2 Concepts and Basic Components

2.1 Concepts and Basic Components, General Information

IndraLogic 2G is a device-independent control programming system.

In accordance with the IEC 61131-3 standard, it supports all standard programming languages but allows only C-routines to be integrated. In combination with the IndraLogic runtime system, it enables several controls project to be programmed in one project.

Observe the following basic concepts that specify programming with IndraLogic 2G:

- **Object orientation:**

  The idea of object orientation is not only expressed in the availability of the corresponding programming elements and functions, but also in the structure and version management of IndraLogic and in the project organization.

  This way, several controls can be arranged in one IndraWorks project.

  The use of various applications on one control is ### in preparation ###.

  Devices that can be parameterized and programmed can be addressed in the same project.

- **IndraLogic versions:**

  IndraLogic is available as IndraLogic 1.x and IndraLogic 2G. Both can be provided with an installation.

  Customers that choose a control that supports IndraLogic 2G can use other controls supporting 2G in the project.

  The same applies for controls supporting IndraLogic 1.x.

  However, a mix of controls, some of which use IndraLogic 1.x and some of which use IndraLogic 2G, is not possible within one IndraWorks project.

☞ Data can be exchanged between a 1.x and a 2G control via network variables, page 71,.

- **Project organization:**

  This is also shaped by the idea of object orientation:

  An IndraLogic project includes a control program consisting of various programming objects and the definition of resources needed to operate instances of this program - handled as "Application" objects - on a specified target system (device, module, control).

  There are two main types of objects in a project:

  1. **Programming objects (POUs):**

     These are programs, functions, function blocks, methods, interfaces, actions, data type definitions, etc. See What is a POU Object, page 27.

     – Programming objects instantiated project-wide, i.e. for all applications defined in the project, have to be managed in the "General module" folder. Instantiation occurs when a program POU is called from the task, page 67, of an application.

     – Programming objects directly assigned to an application cannot be instantiated by other applications.

2. **Resource objects:**

   These are device objects, applications, task configurations, recipe managers, etc.

   According to the rules that apply when creating device objects in the Project Explorer, the hardware environment has to be mapped. See Devices in the Project Explorer, page 63.

   The validity range of objects such as "libraries" and "global variable lists" are defined hierarchically, e.g. by arranging application and device objects.

- **Code generation:**

  Code generation using integrated compilers and machine code allows short execution times.

- **Data transfer to the control device:**

  Data is transferred between IndraLogic 2G and the control using a gateway (component) and a runtime system. A complete online functionality for monitoring the program is available on the control.

2.2 Differences from IndraLogic 1.x

In principle, projects created with IndraLogic 1.x can be opened and processed.

IndraLogic 2G provides the following extensions and improvements:

Object orientation

Object orientation at programming and in the project structure

- *Extensions for function blocks:*
  - Properties, page 46
  - Interfaces, page 49
  - Methods, page 45
  - Inheritance, page 36,
  - Method call, page 40
- Extendable functions, page 33 ### in preparation ###
- Device-dependent applications, page 66, as instances of independent programming objects

New with regard to data types

- ANY_TYPE, ### In preparation ###,
- UNION, page 564
- LTIME, page 555
- REFERENCE, page 556
- Enumeration, page 564, basic data type can be specified
- `di : DINT := DINT#16#FFFFFFFF;` not permitted.

Detailed information under Data Types, page 552,.

New with regard to operators and variables

- New validity range operators, page 608, extended namespaces
- Pointers, page 557, replace the `INSTANCE_OF` operator
- Init methods, page 524, replace the INI operator
- Exit methods, page 45
- Output variables in function calls, page 31, and method calls, page 40
- VAR_TEMP, page 518, VAR_STAT, page 518

- Any expression, page 510, for variable initialization
- Assignment, page 389, as expression
- Index access, page 557, for pointers and strings
- Extendable functions (variable number of parameters) - ### in preparation ###.

New visualization concept

- A visualization editor is provided. This editor works with a toolbox and with an editor for the element properties. Parts of the visualization functionality are implemented according to IEC 61131 and thus - like the visualization elements - provided in the libraries. An internal runtime system executes the most important visualization functions.
- Text lists and image pools

New concept for user management and security

- User accounts, user groups, group-dependent rights for access to and actions with individual objects

News in editors

- ST editor:

  Parenthesis, breaks, code completion, Inline monitoring, Inline set/reset assignment
- IL editor as table editor
- FBD, LD and IL are mutually convertible and have a common editor
- FBD/LD/IL editor: The main output can be set in function blocks with multiple outputs (### in preparation ###)
- FBD/LD/IL editor: Function block parameters are not automatically updated
- FBD/LD/IL editor: Branching and "networks in networks"
- SFC editor: Only one step type, macros, multiple selection of independent elements, no syntax check while editing

News with regard to library management

- Several versions of one library are possible in the same project. Unique access by specifying the namespace.
- Installation in repositories, automatic updates, debugging

And more...

- Configurable menus, toolbar and keyboard operation
- Control configuration and task configuration integrated in the Project Explorer
- Unicode support
- Single line comments: `// Comment`
- `CONTINUE` in loops
- Multiple selection in the Project Explorer
- Online help integrated in the user interface
- Conditional compilation
- Conditional breakpoints
- Debugging: Execution up to cursor, execution up to return
- Field bus driver according to IEC 61131-3
- Symbol and control configuration in the application
- Free memory assignment for code and data
- Each object can be defined as "internal" or "external" (late linking in the runtime system).
- Connection to external data sources

Compatibility with IndraLogic 1.x projects

- Projects in other formats including projects created with IndraLogic 1.x can be imported. The handling of integrated libraries and devices can be specified here.
- Syntactic and semantic limitations with regard to IndraLogic 1.x projects:
  - `FUNCTIONBLOCK` is no longer a valid keyword; but has been replaced by FUNCTION_BLOCK, page 33,.
  - TYPE, page 563, (structure declaration) has to be followed by a `":"`.
  - ARRAY initialization, page 560, requires square brackets.
  - Local declaration of an enumeration, page 564, is now only possible within `TYPE` / `END_TYPE`.
  - `INI` is no longer supported (replaced by the Init method, page 524,).
  - In function calls, page 31. it is no longer possible to mix explicit and implicit assignments. However, this allows the arrangement of the input parameters to be modified:
- Pragmas, page 526, (import of IndraLogic 1.x pragmas ### in preparation ###)

2.3 Project

A project includes the POU objects, page 27, that compose a control program and the definitions of the resource objects, page 63 needed to execute one or more instances of the program (application, page 66) on a specified target system (controls, devices).

POU objects available project-wide are managed in the "General module" folder. Device-specific resource objects and application-specific POU objects are managed in the respective "device".

A project is saved in an "IndraLogic.project" file.

Project-specific configurations can be made in the dialogs of the **Tools ▸ Settings** and **Tools ▸ Options** menu items.

> ☞ The appearance and properties of the user interface are saved in the programming system and not with the project.

Library information

Information on the project currently edited, e.g. file data, object statistics, the name of the author name, etc., is found in the "Library Information" dialogs.

The library information, page 371, is provided in the "Library Info" object in the "General module" folder.

Data transfer

Devices and individual device objects can be imported into a project. In IndraLogic V1.x and in IndraLogic 2G projects can be imported.

If libraries or devices are integrated in a V1.x project, decide before converting the project whether they should continue to be used in the project or should be replaced by others or whether references should be removed.

Also see

Data transfer, page 115.

2.4 Supported Programming Languages

All programming languages listed in the IEC standard IEC 61131 are supported with specially adapted editors:

- FBD/LD/IL editor, page 339, for function block diagrams (FBD), ladder diagrams (LD) and instruction lists (IL)
- SFC editor, page 399, for sequential function chart (SFC)
- ST editor, page 386, for structured text
- In addition, an editor is provided that supports programming in the CFC (continuous function chart) language:

  CFC editor, page 309, stands for Continuous Function Chart editor.

  CFC is an extension of the standard IEC languages.

2.5 What is a POU Object

POUs are programming units (objects) composing a control program.

POU	= Program organization unit
POE	= Program organization unit

POUs managed in the "General module" folder are not device-specific. Instead, they can be used project-wide and can be instantiated for usage in a device-specific application. For this purpose, program POUs are called using a task of the respective application.

POUs assigned to a "device", i.e. are added to the Project Explorer, page 63, directly below an application can only be used by applications listed below this application in an indented list ("child" applications). Further information can be found in the description of the project tree, page 65, and the "Application" object, page 66.

Using Add, page 234, in the library and in the context menu, "POU" is also used as the name for a specific Subcategory, page 28, of these objects and in this case it only designates programs, function blocks and functions.

A "Program Organization Unit" object is always a programming unit, an object that can be managed in a non-device-specifically in the "General module" folder or in a device-specifically below an application and is displayed in an editor window and can be edited there. A POU object can be a program, a function, a function block, a method, an action, an interface or a DUT (data unit type).

Note that it is possible to define specific properties, page 238, (such as specifications for the compilation, etc.) separately for each POU.

Types of POUs:

- Action, page 51
- Application, page 66
- Library manager, page 367
- Image pool, page 61
- DUT (data type), page 43
- Property (PROPERTY), page 46
- Function (FUNCTION), page 31
- Function block (FUNCTION_BLOCK), page 33
- Global variable list, page 52
- Method (METHOD), page 45
- Program (PROGRAM), page 29
- Interface (INTERFACE), page 49
- Text list, page 55

- Visualization, page 63

In addition to POU objects, "resources" are needed to execute the program on the target system (application, task configuration, etc.). These are managed below a "device" in the Project Explorer, page 63,.

2.6 Program Organization Units (POU)

2.6.1 POU, General Information

The term "POU" designates a program organization unit that is either of type program, function or function block. For superordinate use of the term "POU" for all program organization units, see "What is a POU Object", page 27,. There is also information on managing project-global and device-specific POUs.

A POU can be added to the project by using **Add ▸ POU** in the context menu.

Alternatively, add a POU object from the "PLC Objects" library by dragging it with the mouse.

The dialog "Add POU" opens in which POUs are configured with regard to name, type and implementation language.

For a function block, EXTENDS and IMPLEMENTS properties can be added.

For a function, a property or a method, a return value has to be specified in most of the cases.

Depending on the type and implementation language used, a function block can be extended by method, properties, actions and transitions.

The hierarchical processing of individual POUs assigned to an application depends on the device-specific configuration (Project Explorer).

Each POU consists of a "declaration part" and an "implementation part". The implementation part is written in one of the following programming languages:

- *Text languages:*
  - Instruction list (IL)
  - Structured text (ST)
- *Graphical languages:*
  - Sequential function chart (SFC), structuring medium
  - Ladder diagram (LD)
  - Function block diagram (FBD)
  - Continuous function chart (CFC)

IndraLogic 2G supports the POUs described in the standard 61131-3. To use these default POUs in your project, page 26,, the standard.library library has to be included.

Calling POUs

POUs can call other POUs. Recursions are not permitted.

If a POU belonging to an application calls another POU only by its name (without namespace, page 86, extension), the following order applies in which it is searched in the project for the POU to be called:

1. Current application,
2. Library manager of the current application,
3. POU window
4. Library manager in the POU window.

If a POU with the name specified in the call exists in a library of the "Application" library manager and as object in the "POUs" window, there is no syntax on how the POU in the "POU s" window can be called by its name only.

In this case, the respective library of the "Application" library manager is to be moved to the library manager in the "POUs" window. The POU object in the "POUs" window can the be called using only its name (and if required, the POU of the library by adding the library namespace).

2.6.2 Program (PROGRAM)

A program is a POU, page 28 that provides one or more values at execution. All values remain from one program execution until the following one.

Adding a program: A program object can be added to the project via **Add** ▸ **POU** in the context menu.

- If it is to be directly assigned to an existing application, the "Application" object, page 66, has to be highlighted before in the Project Explorer.
- If it is to be available across the projects, it has to be added to the "General module" folder.

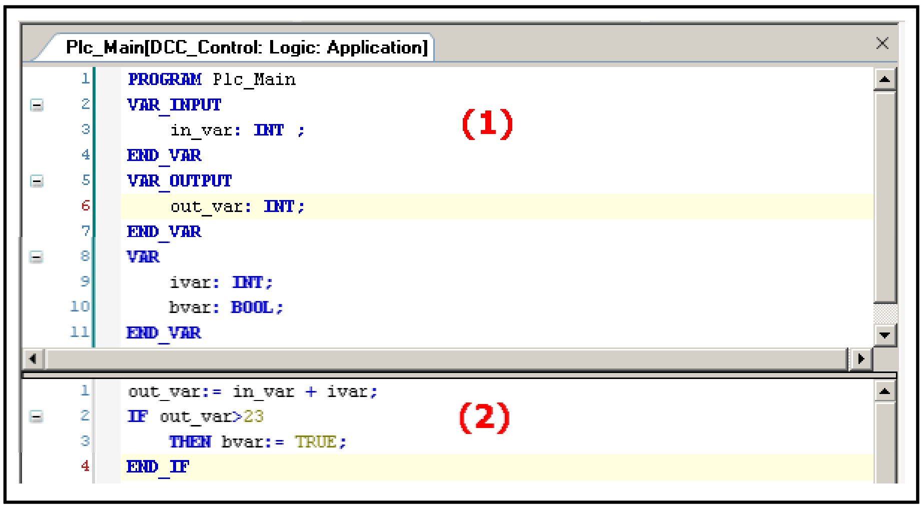
In the "Add Object" dialog, select the POU type "Program", enter a name for the program (<program name>) and select the desired implementation language (programming language). After the settings have been confirmed with "Finish", the new POU object is displayed in the Project Explorer. Open the editor window for the new program by double-clicking the POU object or via the "Open" command in the context menu and then start the implementation:

Declaration: *Syntax:*

```
PROGRAM <program name>
```

The variable declarations for

- Input Variables (VAR_INPUT), page 517,
- Output Variables (VAR_OUTPUT), page 517,
- Local Variables (VAR), page 516,
- External Variables (VAR_EXTERNAL) , page 519, and
- Access Variables, page 519 (### In preparation ###).



```
PROGRAM Plc_Main
VAR_INPUT
    in_var: INT ;
END_VAR
VAR_OUTPUT
    out_var: INT;
END_VAR
VAR
    ivar: INT;
    bvar: BOOL;
END_VAR
```

```
out_var:= in_var + ivar;
IF out_var>23
    THEN bvar:= TRUE;
END_IF
```

(1) Declaration
(2) Implementation
Fig.2-1: Program example

Program calls:

A program can be called by another program or function block instance.

But

- a program call in a function, page 31, is not permitted.
- There are no instances of programs.

If a program was called and that caused changes in the program values, these changes remain until the program is called again. This is also the case if the new call is made by another program or another function block instance.

This differs from calling a function block where only the values in the respective instance of the function block change and the changes are only to be noted if the same instance is called again.

To set input and output parameters at the program call, use a parenthesis right after the program name.

For input parameters, the assignment is given with ":=" as for the initialization, page 509, of variables in the declaration.

For output parameters "=>" is used; see the example below.

If a program call is added by using the "Input assistance" and the option "Add with arguments" in the implementation part of a text editor, it is automatically displayed with all parameters according to the syntax described in the previous section.

Then, add the corresponding value assignments.

To enable the option "Add with arguments", right-click in the editor workspace and select **Input assistance** in the context menu. In the dialog, set the option "Add with arguments".

Program call examples:

In IL (instruction list):

```
CAL                 PRG_example(
           in_var:= 33)
LD                  PRG_example.in_var
ST                  erg
```

or with parameter assignment (input assistance "Add with arguments", see above):

```
CAL                 PRG_example(
           in_var:= 33,
          out_var=> erg)
```

In ST (structured text):

Note that - in contrast to IndraLogic 2.x - parentheses are required here!

Program:

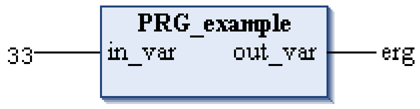
```
PRG_example.in_var:= 33;
PRG_example();
erg:= PRG_example.out_var;
```

or better, with parameter assignment (input assistance "Add with arguments", see above):

Program:

```
PRG_example(in_var:= 33, out_var=> erg);
```

In FBD (function block diagram):



2.6.3 Function (FUNCTION)

A function is a POU, page 28 that can be exactly one data element (can also consist of multiple elements as array or structure for example) at execution. In addition to the return value, output variables can also be provided.

The call, page 32, can be an operator in expressions in text languages.

Add function: A function object can be added to the project via the context menu items **Add ▸ POU**.

- If it is to be directly assigned to an existing application, the "Application" object, page 66, has to be highlighted before in the Project Explorer.
- If it is to be available across the projects, it has to be added to the "General module" folder.

In the "Add Object" dialog, select the POU type "Function", enter a name for the function (<function name>), select a return type (<data type>) and select the desired implementation language (programming language).

Use the ... button to open the input assistance, page 98, to select the return type.

After the settings have been confirmed with "Finish", the new POU object is displayed in the Project Explorer. Open the editor window for the function object by double-clicking the POU object or via "Open" in the context menu and then start the implementation:

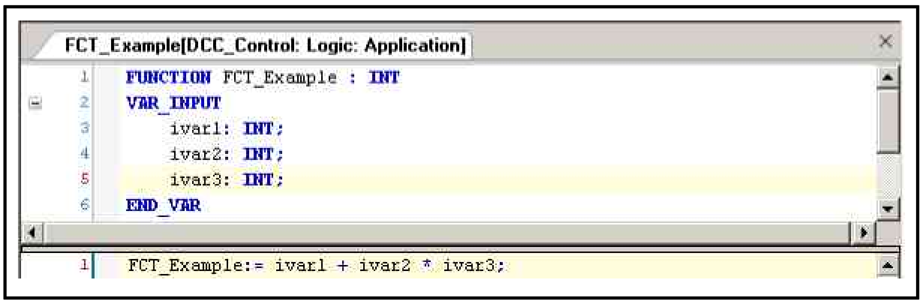
Declaration: *Syntax:*

```
FUNCTION <function name>: <data type>
```

The variable declarations for

- Input Variables (VAR_INPUT), page 517,
- Local Variables (VAR), page 516, and if required
- Output Variables (VAR_OUTPUT), page 517.

A result has to be assigned to each function (return value with type and name of the function).



```
FUNCTION FCT_Example : INT
VAR_INPUT
    ivar1: INT;
    ivar2: INT;
    ivar3: INT;
END_VAR
```

```
FCT_Example:= ivar1 + ivar2 * ivar3;
```

Fig.2-2: Example: Function in ST (The function has three input variables and returns the product of the last two added to the first.)

☞ If a local variable is declared in a function as RETAIN, it has no effect!

The variable is not written in the retain memory!

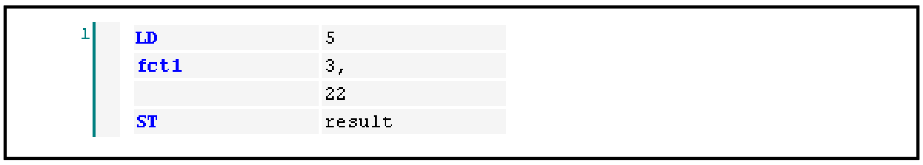
☞ The "sequential function chart" (SFC) structuring tool is not intended for functions.

Function call: In "ST", page 389, (structured text), a function call can be used as an operand in expressions.

In "SFC" (sequential function chart), a function call can only occur in step actions or transitions.

In contrast to programs or function blocks, function variables are reassigned for each call. This means that function calls with the same arguments (input parameters) always return the same value (return value/output parameter). For this reason, functions may not use global variables and addresses!

Examples of function calls:



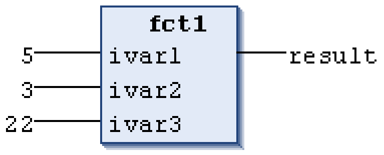
```
LD      5
fct1    3,
        22
ST      result
```

Fig.2-3: In IL (instruction list):

In ST (structured text):

```
result:= fct1(5, 3, 22);
```

In FBD (function block diagram):



☞ In contrast to IndraLogic 1.x, explicit and implicit parameter assignments can no longer be mixed in function calls. Thus, the sequence of parameter assignments at the call is no longer specified.

Example:

```
fun(formal1:= actual1, actual2); // -> Error message
fun(formal2:= actual2, formal1:= actual1);
// Same semantics as the following:
fun(formal1:= actual1, formal2:= actual2);
```

This difference in handling the parameter assignments has to be considered when editing V1.x projects!

VAR_OUTPUT in functions According to the IEC 61131-3 standard, functions can have additional outputs.

Syntax:

```
out1=><Ausgabevariable1>|,out2=><Ausgabevariable2>|,...
```

Example:

The function "fun" is defined with two input variables "in1" and "in2" (VAR_INPUT) and two output values "out1" and "out2" (VAR_OUTPUT).

Function call

```
fun(in1:=1, in2:=2, out1=>loc1, out2=>loc2);
```

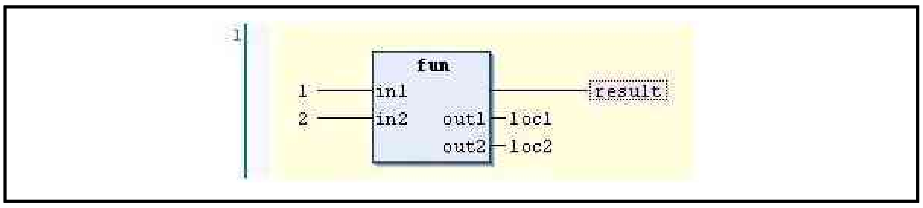


Fig.2-4: Function with return value and two output variables

Extendable functions

As an extension of the IEC 61131-3 standard, functions and methods can be provided with a variable number of input parameters of the same type. Further information can be found in Extendable functions, page 524,.

2.6.4 Function Block (FUNCTION_BLOCK)

Function Block, General Information

A function block is a POU, page 28 providing one or multiple values at execution.

In contrast to a function, the values of the output variables and the local variables used are retained from one execution to the next. This way, when a function block with the same input parameters instance is called more than once, the same output values are not necessarily provided.

Function blocks can be defined as extensions, page 36, of other function blocks and can contain interface definitions, page 38, for method calls, page 40,.

This means that the principles of object-oriented programming (instance generation) of inheritance can be applied when programming with function blocks.

A function block is always called using an instance, page 34.

Adding a function block:

A function block can be added to the project via **Add ▸ POU** in the context menu.

- If it is to be directly assigned to an existing application, the "Application" object, page 66, has to be highlighted before in the Project Explorer.
- To be available project-wide, add it to the "General module" folder.

In the "Add Object" dialog, select the POU type "Function block", enter a name for the function block (<functionblockname>) and select the desired implementation language (programming language).

The following options can also be enabled:

- **EXTENDS (extended):** Enter the name of another function block from the project that is to be used as basis for the present function block.

  A detailed description can be found in the following, seeExtending a Function Block, page 36.
- **IMPLEMENTS (implemented):** Enter the name or names of the interfaces, page 49, defined in the project to be implemented in the present function block.

  Separate multiple interface names with commas.

A detailed description can be found in Implementing Interfaces, page 38.

In the "Method implementation language" field, select the desired programming language for all method objects created by the interface implementation. This is not related to the set function block programming language.

After the settings have been confirmed with "Finish", the editor window for the new function block opens. Start the implementation.

Declaration

Syntax:

```
FUNCTION_BLOCK <function_block type name>
               | EXTENDS <function_block type name>
               | IMPLEMENTS <comma-separated list of interface names>
```

The variable declarations for

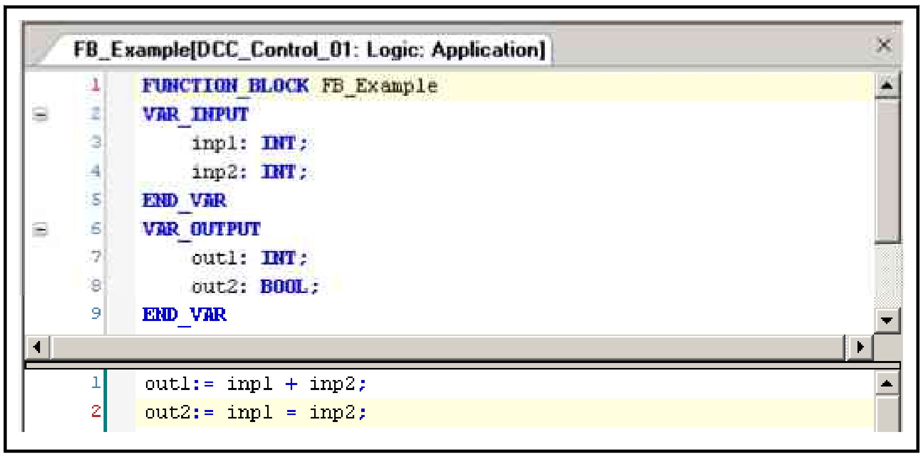
- Input Variables (VAR_INPUT), page 517,
- Output Variables (VAR_OUTPUT), page 517,
- Local Variables (VAR), page 516,
- External Variables (VAR_EXTERNAL) , page 519, and
- Access Variables, page 519 (### In preparation ###).

Example:

The function block example in the figure below has two input variables "inp1" and "inp2" and two output variables "out1" and "out2".

"out1" is the sum of both inputs.

"out2" is the result of an equality check.


FB_Example[DCC_Control_01: Logic: Application]

```
FUNCTION_BLOCK FB_Example
VAR_INPUT
    inp1: INT;
    inp2: INT;
END_VAR
VAR_OUTPUT
    out1: INT;
    out2: BOOL;
END_VAR
```

```
out1:= inp1 + inp2;
out2:= inp1 = inp2;
```

Fig.2-5: Example of a function block in ST

In contrast to IndraLogic 1.x, "FUNCTIONBLOCK" is no longer a valid keyword.

It has to be replaced by **FUNCTION_BLOCK**!

Instance of a Function Block

Function blocks, page 33, are always called using an instance (copy of the function block).

Each instance has an identifier (instance name) and a data structure that contains its input, output and internal variables.

Like variables, instances are declared to be local or global where the name of the function block is specified as type of the identifier.

Syntax:

```
<Instance name>: <function_block name>;
```

Example:

Declaration (e.g. in the declaration part of a program) of an instance "INSTANCE" of the function block "FUB":

```
INSTANCE: FUB;
```

☞ Instance declarations are typically made in the declaration parts of function blocks and programs; range VAR...END_VAR or VAR RETAIN...END_VAR or for the data transfer in the range of VAR_INPUT...END_VAR.

In functions, they are only possible for the data transfer in the range VAR_INPUT...END_VAR.

Calling a Function Block

Function blocks, page 33, are always called using a function block instance. The instance has to be declared as local or global (<InstanceName>).

This declaration is explained in Instance of a function block, page 34.

The desired function block variable (<VariableName>) can be accessed using the following syntax:

Syntax of calling an input variable:

```
<Instance name>.<Variable name>
```

Note the following:

- From outside the function block instance, only the function block input and output variables can be accessed:
  - From the outside, input variables can only be described
  - From the outside, output variables can only be read
  - From the outside, local variables are not visible
- Access to a function block instance is limited to the POU, page 28, in which it is declared unless it is declared as global.
- When calling the instance, the desired values can be assigned to the function block parameters.

  See below "Assigning Parameters at a Call", page 36.
- Input/Output Variables (`VAR_IN_OUT`) of a function block are transferred as pointers.
- In SFC, function block calls can only occur in actions.
- The name of a function block instance can be used as an input parameter for a function or another function block.
- All values of a function block remain until the next function block execution. For this reason, function block calls do not always deliver the same output values, even if the same input values are used!

☞ If at least one of the function block variable is a "remanent" variable, the entire instance is saved in the retain area.

Examples for accesses to function block variables:

Assumption: Function block **fb** has an input variable "in1" of type `INT`.

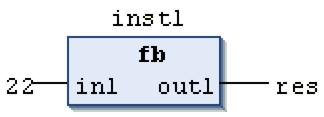
The following shows the calls of this variable from the program "prog".

Declaration and implementation in ST:

```
PROGRAM prog
 VAR
  inst1:fb;
 END_VAR

inst1.in1:= 22;  // the value 22 is assigned to the input variable in1 of the instance inst1
inst1();         // fb is called and edited; this is necessary for the following access
                 // to the output variable
res:=inst1.outl; // the output variable out1 of fb is read
```

Example in FBD (function block language):



Assigning parameters in a call:

In the text languages "IL" and "ST", input and output parameters can be set directly when calling the function block. The values can be assigned to the parameters within parentheses directly following the function block name.

For input parameters, the assignment is given with ":=" as for the initialization, page 509, of variables in the declaration.

For output parameters "=>" is used; see the following example.

Example, call with assignments:

In this example, a timer function block (instance "CMD_TMR") with assignments for the "IN" and "PT" parameters is called. Afterwards, the output variable "Q" of the timer is assigned to variable "A".

Syntax of calling an output variable:

```
<Instance name>.<Variable name>
```

Example:

```
 CMD_TMR(IN := %IX5, PT := 300);
 A := CMD_TMR.Q
```

If the instance is added to the programming section of a text editor using input assistance and the "Add with arguments" option, it is automatically represented according to the syntax, page 36, described above with all parameters. Then, add the corresponding value assignments.

To enable the option "Add with arguments", right-click in the editor workspace and select **Input assistance** in the context menu. In the "input assistance" dialog, place a checkmark next to the "Add with arguments" option.

For the example described above, the call would then be as follows:

Example, adding with arguments using input assistance:

```
 CMD_TMR(IN := %IX5, PT := 300, Q=>A);
```

Extending a Function Block (EXTENDS)

In object-oriented programming, one function block, page 33, can be derived from another function block. That means that one function block can be used to extend another thereby adding the properties of the basic function block to its own.

This extension is carried out in the declaration using the keyword **EXTENDS**.

The "Extended" option can be enabled when a function block is added to the project. To do this, highlight the "Application" node and select the "Add Object" dialog in the **Add ▸ POU** context menu.

Syntax:

```
FUNCTION_BLOCK <function_block type name B> EXTENDS <function_block type name A>
```

See the variable declarations in the following.

Definition of function block fbA:

```
FUNCTION_BLOCK fbA
VAR_INPUT
    ivar_A: int;
.....
```

Definition of function block fbB:

```
FUNCTION_BLOCK fbB EXTENDS fbA
VAR_INPUT
    ivar_B: int;
.....
```

Extending using EXTENDS means that:

- "fbB" contains all data and methods/properties defined by "fbA"; has"ivar_A" (inherited) and "ivar_B" (itself) as VAR_INPUT.

  An instance of "fbB" can now be used in every context in which a function block of type "fbA" is expected.
- "fbB" may overwrite the methods/properties defined in "fbA".

  That means that "fbB" can define a method/property with the same name, the same inputs and the same return value (as well as outputs if available) as defined by "fbA".

  If it does not overwrite the method/properties, it inherits the original.
- "fbB" may not contain any function block variables with the same name as those used in "fbA". If this is the case, the compiler reports an error.

  The only exception:

  If a variable is declared in "fbA" as **VAR_TEMP**, "fbB" may define a variable of the same name, but can no longer access the variable of the basic function block.
- "fbA" methods and variables can be directly addressed within the valid range of "fbB" by using the keyword **SUPER**

  (`SUPER^.<MethodName>` or `SUPER^.<MethodName>.<Variable-Name>`)

☞ Multiple inheritance is not permitted!

Example:

```
FUNCTION_BLOCK FB_Base
VAR_INPUT
END_VAR
VAR_OUTPUT
    iCnt : INT;
    iRes : INT;
END_VAR
```

```
VAR
END_VAR

THIS^.METH_DoIt();
THIS^.METH_DoAlso();

    METHOD METH_DoIt : BOOL
    VAR
    END_VAR
    iCnt := -1;
    METH_DoIt := TRUE;

    METHOD METH_DoAlso : BOOL
    VAR
    END_VAR
    iRes := -5;
    METH_DoAlso := TRUE;

FUNCTION_BLOCK FB_1 EXTENDS FB_Base
VAR_INPUT
END_VAR
VAR_OUTPUT
    iCnt_Base: INT;
    iCnt_THIS: INT;
    iRes_Base: INT;
    iRes_THIS: INT;
END_VAR
VAR
END_VAR
// Calls the method defined under FB_1
 THIS^.METH_DoIt();
 THIS^.METH_DoAlso();
 iCnt_THIS:= iCnt;
 iRes_THIS:= iRes;
// Calls the method defined under FB_Base
 SUPER^.METH_DoIt();
 SUPER^.METH_DoAlso();
 iCnt_Base:= iCnt;
 iRes_Base:= iRes;

    METHOD METH_DoIt : BOOL
    VAR
    END_VAR
      iCnt := 1111;
      METH_DoIt := TRUE;

PROGRAM PLC_PRG
VAR
    Myfb_1: FB_1;
    iFB: INT;
    iBase: INT;
END_VAR
 Myfb_1();
 iBase := Myfb_1.iCnt_Base;
 iFB := Myfb_1.iCnt_THIS;
```

Implementing Interfaces (IMPLEMENTS)

In object-oriented programming, a function block can implement interfaces, page 49, that enable the use of methods, page 45, and properties, page 46,:

Syntax:

```
FUNCTION_BLOCK <FB type name> IMPLEMENTS <interface name_1>,...,<interface name_n>
```

A function block that implements an interface has to contain all methods/ properties defined in this interface. That means that the name, the inputs and return values (as well as outputs if available) of the methods/properties have to be identical.

In addition, when a new function block that implements an interface is created, all methods and properties defined in the interface are also automatically copied below the new function block.

Currently, changes made later on at the interface, such as adding more methods, are not automatically made in the respective function blocks (interface methods become POU methods in function blocks, interface properties become POU properties in function blocks).

This has still to be carried out explicitly using the Implement interfaces, page 236, at each function block. The implementation language is queried for the method/property.

Example:

INTERFACE ITF_1 contains the method "GetName":

```
METHOD GetName : STRING
```

The function blocks FB_A and FB_B implement the interface ITF_1 each:

```
FUNCTION_BLOCK FB_A IMPLEMENTS ITF_1
END_FUNCTION_BLOCK

FUNCTION_BLOCK FB_B IMPLEMENTS ITF_1
END_FUNCTION_BLOCK
```

This means that the method "GetName" has to be present in both functions and is automatically attached below in the Project Explorer when the function blocks are created. It has to be implemented separately for each function block.

Function blocks FB_A, method "GetName" (example)

```
METHOD GetName : STRING
 GetName:= 'FB_A';
```

Function blocks FB_B, method "GetName" (example)

```
METHOD GetName : STRING
 GetName:= 'FB_B';
```

Look at the "DeliverName" function with its input of a variable of type of the interface ITF_1:

Function "DeliverName": STRING

```
FUNCTION DeliverName : STRING
VAR_INPUT
   I_i: ITF_1;
END_VAR
 DeliverName:= I_i.GetName();  // in the case, it depends on the "actual" type of I_i,
                               // if A.GetName or B.GetName is called.
```

This input variable can receive all function blocks that implement "interface ITF_1".

☞ The interface of a function block has to be assigned to the variable of the type of an interface before it can be used to call a method.

The variable of an interface type is always a reference of the assigned function block instance.

In this way, calling the interface method results in a call of the function block implementation.

In online mode, as soon as a reference is created, the related address is shown.

If a reference has not yet been generated, the value "0" is shown here.

Examples for function calls:

```
PROGRAM PlcProg
VAR
    Name_FB_A : STRING;
    Name_FB_B : STRING;
    FB_A_Inst: FB_A;
    FB_B_Inst: FB_B;
END_VAR
 Name_FB_A:= DeliverName(I_i:= FB_A_Inst); // call with FB_A_instance
 Name_FB_B:= DeliverName(I_i:= FB_B_Inst); // call with FB_B_instance
```

Method Call

Object-oriented programming with function blocks is supported - apart from the option of extension with EXTENDS, page 36 - also by using

Interfaces, page 38, and

Inheritance, page 33,

This requires dynamically resolved method calls, also called "virtual function calls" for further methods.

Virtual function calls require a little more time than normal function calls and are used when:

- a function block is called using a pointer (e.g. "pfub^.method"),
- a method of an interface variable is called (e.g. "interface1.method"),
- a method calls another method of the same function block,
- a function block is called using a reference,
- VAR_IN_OUT of a basic function block type is assigned to an instance of a derived function block type.

Virtual function calls enable the same call in a program source code to call a variety of methods at runtime.

For further information, refer to:

- Method, page 45, about using methods
- THIS pointer, page 42, about using THIS
- SUPER pointer, page 41, about using SUPER.

Calling methods

According to the IEC 61131-3 standard, methods can have additional outputs like functions. These have to be assigned according to the following syntax at method call:

Syntax:

```
<method>(in1:=<value> |, further input assignments, out1 => <output variable 1>
                                              | out2 => <output variable 2> |...further output variables)
```

This causes the method output as defined in the call to be written to the locally declared output variables.

Example:

Assumption:

Function blocks "fub1" and "fub2"

EXTENDS (extend) function block "fubbase"

IMPLEMENT (implement) interface "interface1"

The method "method1" is included.

Possible use of the interfaces and calling the method:

```
// Declaration
VAR_INPUT
```

```
    b : BOOL;
END_VAR
VAR
    pInst : POINTER TO fubbase;
    instBase : fubbase;
    inst1 : fub1;
    inst2 : fub2;
    interfaceRef : interface1;
END_VAR

// Implementation
IF b THEN
    interfaceRef := inst1;    // Interface1 for fub1
    pInst := ADR(instBase);
ELSE
    interfaceRef := inst2;    // Interface1 for fub2
    pInst := ADR(inst1);
END_IF

pInst^.method1();       // If b is true, fubbase.method1 is called, else fub1.method1 is called
interfaceRef.method1(); // If b is true, fub1.method1 is called, else fub2.method1 is called
```

Assuming that "fubbase" contains two methods, "method1" and "method2", and that "fub1" overwrites "method2", but not "method1":

"method1" is called in the following as in the example above:

```
pInst^.method1(); // If b is true fubbase.method1 is called, else fub1.method1 is called
```

SUPER Pointer

Also see the call via THIS pointer, page 42,.

One pointer with the name SUPER is automatically available for each function block. This pointer points to the basic function block instance from which the function block was created by inheritance of the basic function block.

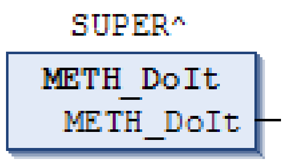
Thus, the following effective problem solution is possible:

- SUPER allows access to the implementation of the basic class methods. Using the keyword SUPER, a method is called that is valid in the instance of the basic or parent class. Thus, no dynamic name linking takes place.

SUPER can only be used in methods and in the respective function block implementations. Since SUPER is a pointer to the basic function block, it has to be unreferenced to keep the address of the function block:

`SUPER^.METH_DoIt.`

Call SUPER in different implementation languages

ST	`SUPER^.METH_DoIt();`
LD/FBD/CFC	
IL	The SUPER functionality for the instruction list (IL) is ### in preparation ###.

Example:

```
FUNCTION_BLOCK FB_Base
VAR_OUTPUT
    iCnt : INT := 5;
END_VAR
    METHOD METH_DoIt : BOOL
    iCnt := -1;
    METH_DoIt := TRUE;
    METHOD METH_DoAlso : BOOL
    METH_DoAlso := TRUE;

FUNCTION_BLOCK FB_1 EXTENDS FB_Base
VAR_OUTPUT
    iSuper: INT;
    iThis: INT;
END_VAR
// Calls the method defined under FB_1
  THIS^.METH_DoIt();
  THIS^.METH_DoAlso();
  iThis := THIS^.iCnt;
// Calls the method defined under FB_Base
  SUPER^.METH_DoIt();
  SUPER^.METH_DoAlso();
  iSuper := SUPER^.iCnt;

    METHOD METH_DoIt : BOOL
    iCnt := 1111;
    METH_DoIt := TRUE;

PROGRAM PLC_PRG
VAR
    myBase: FB_Base;
    myFB_1: FB_1;
    iTHIS: INT;
    iBase: INT;
END_VAR
 myBase();
 iBase := myBase.iCnt;
 myFB_1();
 iTHIS := myFB_1.iCnt;
```

THIS Pointer

A pointer with the name THIS is automatically available for each function block. This pointer points to the function block instance.

Thus, the following effective problem solutions are possible:

- If a locally declared variable shadows a function block variable in the method.
- If the pointer is referenced to the individual function block instance to be used in a function.

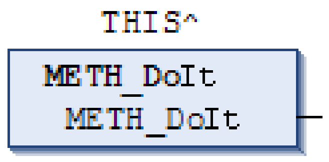
Thus, THIS can only be used in methods and in the respective function block implementations.

THIS has to be written in upper case letters. Other spelling is not permitted.

Since THIS is a pointer to the function block to be inherited, it has to be referenced to keep the address of the overwriting function.

`THIS^.METH_DoIt.`

Call THIS in different implementation languages

ST	`THIS^.METH_DoIt();`
LD/FBD/CFC	
IL	The THIS functionality for the instruction list is (IL) ### in preparation ###.

Example:

The local variable "iVarB" shadows the function block variable "iVarB":

```
FUNCTION_BLOCK  fbA
VAR_INPUT
    iVarA: INT;
END_VAR
 iVarA := 1;
FUNCTION_BLOCK fbB EXTENDS fbA
VAR_INPUT
    iVarB: INT := 0;
END_VAR
 iVarA := 11;
 iVarB := 2;

    METHOD DoIt : BOOL
    VAR_INPUT
    END_VAR
    VAR
        iVarB: INT;
    END_VAR
     iVarB := 22;        // Here the local iVarB is set.
     THIS^.iVarB := 222; // Here the function block variable iVarB is set,
                         // although iVarB is overloaded.

PROGRAM PLC_PRG
VAR
    MyfbB: fbB;
END_VAR

 MyfbB(iVarA:=0 , iVarB:= 0);
 MyfbB.DoIt();
```

Example:

A function call requires the reference to the individual instance:

```
FUNCTION funA : BOOL
VAR_INPUT
    pFB: fbA;
END_VAR
...;

FUNCTION_BLOCK  fbA
VAR_INPUT
    iVarA: INT;
END_VAR
...;

FUNCTION_BLOCK fbB EXTENDS fbA
VAR_INPUT
    iVarB: INT := 0;
END_VAR
 iVarA := 11;
 iVarB := 2;

    METHOD DoIt : BOOL
    VAR_INPUT
    END_VAR
    VAR
        iVarB: INT;
    END_VAR
     iVarB := 22;        // Here the local iVarB is set.
     funA(pFB := THIS^); // Here funA is called with THIS^.

PROGRAM PLC_PRG
VAR
    MyfbB: fbB;
END_VAR
 MyfbB(iVarA:=0 , iVarB:= 0);
 MyfbB.DoIt();
```

2.6.5 DUT/Data Type

In addition to the standard data types, users can define their own data types.

- "Arrays", page 560, (ARRAY),
- "Structures", page 563, (STRUCT),
- "Enumeration types", page 564, (ENUM),

- "References", page 556, (REFERENCE TO),
- "Subrange Types", page 566,
- "Unions", page 564, (UNION)

can be created as data type objects (DUT objects) in the DUT editor, page 338,.

A description of the individual standard and user-defined data types can be found in Data types, page 552.

Adding data type: The "DUT" object can be added to the project via **Add ▸ Data type** in the context menu.

If it is to be directly assigned to an existing application, the application object has to be highlighted before in the Project Explorer.

If the "DUT" object is to be available project-wide, the "General module" folder has to be highlighted. Alternatively, use the mouse to drag the "DUT" object from the "PLC Objects" library to the desired position.

In the "Add Object" dialog, select a name for the new data type from (<DUT name>).

Apply the principle of inheritance for **object-oriented programming** using data types. In the "Add Object" dialog, specify whether the data type is to extend another data type that is already defined in the project. This means that the definitions of extended DUT objects automatically apply here. The extension is enabled via the "Advanced:" option and the name of the "DUT" to be extended is entered.

After the settings have been confirmed with "Finish", the "DUT" object is created in the Project Explorer. Start programming by double-clicking the "DUT" object or via **Open** in the context menu.

Declaration:

☞ The component declaration depends on the type selected, e.g. a structure, page 563, union, page 564, or enumeration, page 564.

Component declaration structure

```
TYPE <DUT name> : <DUT component declaration>
END_TYPE
```

Example:

The following shows two DUT objects defining the structures "struct1" and "struct2"; "struct2" extends "struct1".

This means that the structure "struct2" is provided with four elements (a: INT and b: BOOL, inherited from "struct1"; c and d are self-declared).

TYPE struct1

```
TYPE struct1 :
STRUCT
    a: INT;
    b: BOOL;
END_STRUCT
END_TYPE
```

TYPE struct2 EXTENDS struct1

```
TYPE struct2 EXTENDS struct1 :
STRUCT
    c: DWORD;
    d: STRING;
END_STRUCT
END_TYPE
```

2.6.6 Method (METHOD)

Object-oriented programming is supported by the possible use of methods that contain a sequence of instructions.

A method is not an independent POU, but has to be assigned to a function block, page 33,.

It can be considered as a function in the instance of the respective function block.

Interfaces, page 49, can be used for project-internal method organization for the object-oriented programming.

In this context, an interface is a collection of method prototypes. That means that a method assigned to an interface only contains a declaration part, not a implementation part. The implementation is made in the function block that implements, page 38, the interface and uses the method.

Advantage: The same method call can be used in all function blocks that implement the same interface. That means that the call can be used for a variety of purposes. Calling a method means knowing the purpose to be achieved. That is, the instructions to be actually executed in detail (implementation) to fulfill the purpose depend on the respective function block.

POU method: the method is assigned to a function block. Apart from its declaration, it is also provided with an implementation.

Interface method: the method is assigned to an interface. It has only its declaration part. If the interface is implemented in a function block, the interface methods are implemented and become POU methods.

Adding methods: The "Method" object can be assigned to a function block via the context menu items **Add** ▸ **POU Method**. Alternatively, use the mouse to drag the "POU method" object from the "PLC Objects" library to the desired position.

In the "Add Object" dialog, enter a name for the method (<MethodName>) and a return type (<ReturnDataType>).

Use the ... button to open the input assistance, page 98, to select the return type.

For a method assigned to a function block (POU method), select an implementation language (programming language). After the settings have been confirmed with "Finish", the editor window for the method opens.

For a method assigned to an interface (interface method), select an implementation language (programming language). The implementation is carried out when implementing the interface in the function block.

Declaration: *Syntax*

```
METHOD <method name> : <return data type>
VAR_INPUT
    x: INT;
END_VAR
```

In "Interfaces", page 49,, there is a description on the definition of interfaces that handle methods.

Method call: Method calls, page 40, are also called "virtual function calls".

☞ All data for a method is volatile data and only apply at the execution of a method (stack variables).

In the implementation part of a method, **access to the function block instance** variables is allowed.

If required, use the THIS pointer, page 42,.

Note that a locally declared variable might overwrite a function block variable.

VAR_IN_OUT or VAR_TEMP function block variables cannot be accessed in a method!

Like functions, methods can obtain additional outputs. These have to be assigned in the method call, page 40,.

Special function block methods:

- FB_init method (see also page 524):

  A method called "FB_init" is always declared implicitly, but can also be declared explicitly. It contains the initialization code for the function block as defined in the function block declaration part.
- FB_init method (see also page 524):

  If a method called "FB_reinit" is declared, it is called if an instance of the function block is copied. It starts a re-initialization of the new instance module.
- FB_exit method (see also page 526):

  If a method called "FB_exit" is declared, it is called for each instance of the function block before a download or at an online change.

  For further information, see Declaration, page 526.
- Properties (see also page 46): Properties.

Calling a method when the application is in STOP state

In the device description, page 63,, it can be defined that a certain method of a certain function block instance (a library function block) is always to be called task cyclically.

If the method contains the following input parameters, it is also processed if the active application is currently in STOP state:

Calling a method..., declaration

```
VAR_INPUT
    pTaskInfo : POINTER TO DWORD;
    pApplicationInfo: POINTER TO _IMPLICIT_APPLICATION_INFO;
END_VAR
```

The application status can be queried using "pApplicationInfo" and the corresponding instructions can be programmed; see the following example.

Calling a method..., implementation

```
IF pApplicationInfo^.state=RUNNING THEN
    <instructions>
END_IF
```

2.6.7 Property (PROPERTY)

The "Property" object (PROPERTY) can be assigned to a function block. To add the property object to the project, highlight the function block and select the context menu items **Add** ▸ **POU property** in the context menu. Alternatively, use the mouse to drag the "POU property" object from the "PLC Objects" library to the desired position.

In the "Add Object" dialog, enter a name for the property (<PropertyName>) and a return type (<ReturnDataType>).

Use the ... button to open the input assistance, page 98, to select the return type.

For a POU property, select an implementation language (programming language). After the settings have been confirmed with "Finish", the "Property" object is created in the Project Explorer.

Start programming by double-clicking the "Property" object or via **Open** in the context menu to open the editor.

Such a "Property" contains two special methods, page 45. They are automatically attached below the "Property" object in the object tree:

- The **Set** method is called if the property is to be accessed by writing, i.e. the name of the property is used as an input parameter.
- The **Get** method is called if the property is to be accessed by reading, i.e. the name of the property is used as an output parameter.

Interfaces, page 49, can be used for the project-internal organization of a property in object-oriented programming.

In this context, an interface is a collection of property prototypes. That means that a property that is assigned to an interface only contains a declaration part, not a implementation part. The implementation is made in the function block that implements, page 38, the interface and uses the property.

Advantage: The same property call can be used in all function blocks that implement the same interface. That means that the call can be used for a variety of purposes. Calling a property means knowing the purpose to be achieved. That is, the instructions to be actually executed in detail (implementation) to fulfill the purpose depend on the respective function block.

POU property: the property is assigned to a function block. Apart from its declaration, it is also provided with an implementation.

Interface property: the property is assigned to an interface. It has only its declaration part. If the interface is implemented in a function block, the interface properties are implemented and become POU properties.

Example:

Function block "FB1" uses a local variable "milli". This variable is determined by the properties "second" using the methods "Get" and "Set":

Get:

```
seconds := milli / 1000;
```

Set:

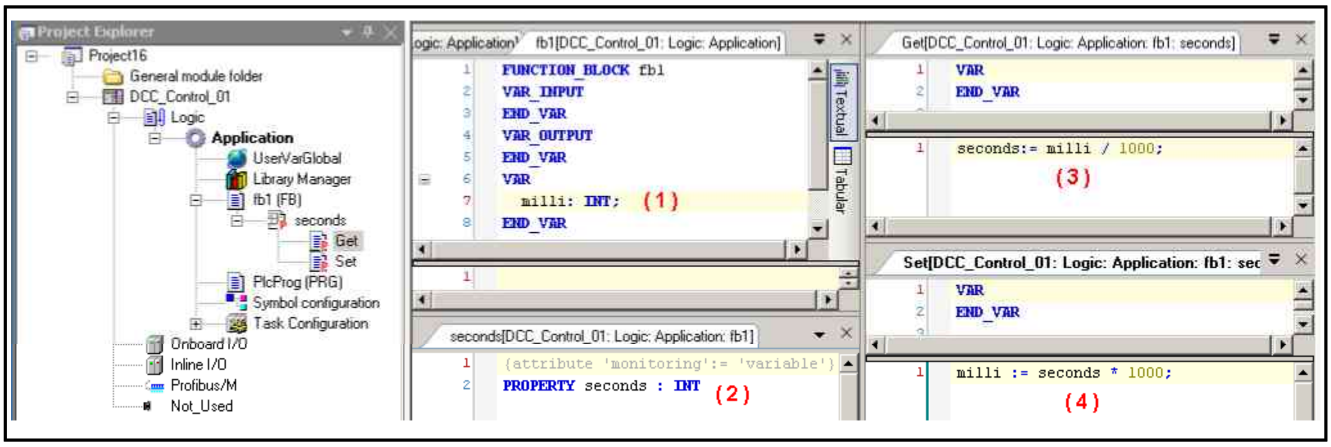
```
milli := seconds * 1000;
```

It can be written on this property (Set method), e.g. with

`fbinst.seconds := 22;` ("fbinst" is the instance of "fb1").

This property can be read (Get method), e.g. with

`testvar := fbinst.seconds;`.



```
FUNCTION_BLOCK fb1
VAR_INPUT
END_VAR
VAR_OUTPUT
END_VAR
VAR
  milli: INT;   (1)
END_VAR
```

```
{attribute 'monitoring':= 'variable'}
PROPERTY seconds : INT   (2)
```

```
VAR
END_VAR
```

```
seconds:= milli / 1000;
(3)
```

```
VAR
END_VAR
```

```
milli := seconds * 1000;
(4)
```

(1)	Function block "fb1" with the local variable 'milli'
(2)	Property 'seconds' with attribute pragma
(3)	"Get" of the 'seconds' property
(4)	"Set" of the 'seconds' property

Fig.2-6: *Example of the 'seconds' property prepared for monitoring*

The figure below shows in the upper part the program with the test variable 'testvar' and the declaration of the function block instance.

The implementation includes in line 1: "Set" method and in line 2: "Get" method.

The figure below shows the monitoring including the display of the 'seconds' property value.

Also refer to

- Attribute monitoring, page 536.

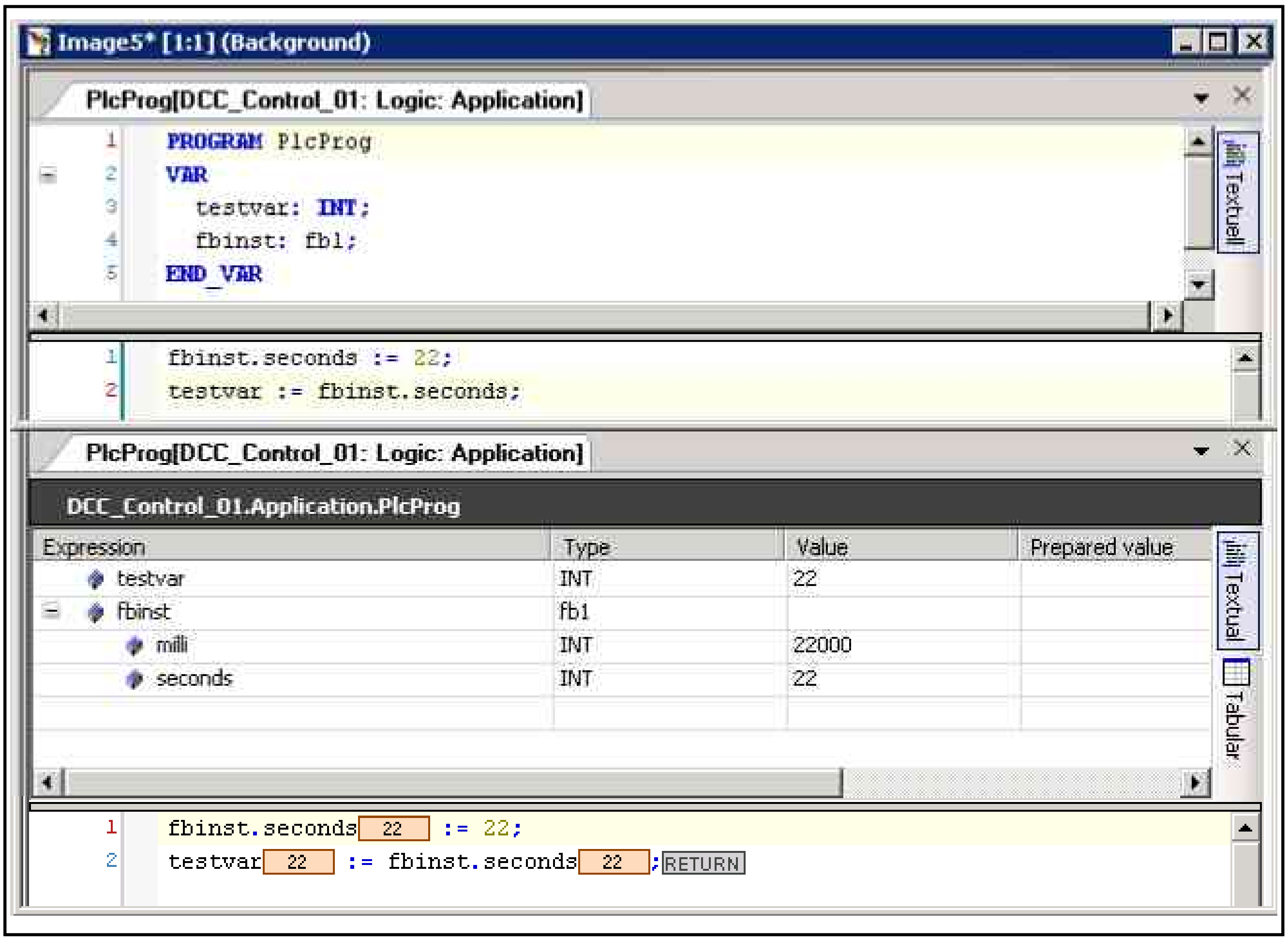


Image5* [1:1] (Background)
PlcProg[DCC_Control_01: Logic: Application]
Textual
Tabular
DCC_Control_01.Application.PlcProg
RETURN

```
PROGRAM PlcProg
VAR
  testvar: INT;
  fbinst: fb1;
END_VAR
```

```
fbinst.seconds := 22;
testvar := fbinst.seconds;
```

Expression	Type	Value	Prepared value
testvar	INT	22	
fbinst	fb1		
milli	INT	22000	
seconds	INT	22	

```
fbinst.seconds[ 22 ] := 22;
testvar[ 22 ] := fbinst.seconds[ 22 ];RETURN
```

Fig.2-7: Example of a monitoring view with the 'seconds' property

A property can also contain local variables, but no additional inputs and - in contrast to a function, page 31, or method, page 45 - no additional outputs.

2.6.8 Interface (INTERFACE)

The use of interfaces is another tool in object-oriented programming.

An "interface" is a POU, page 27 that describes a collection of method prototypes (Interface methods, page 45 / Interface properties, page 46).

Interfaces can be used to manage method prototypes implemented by function blocks, page 33,.

"Prototype" means that only declarations are included, but no implementation.

A function block can implement, page 38, an interface which means that basically all method prototypes specified in an interface have to be available in the function block.

Advantage:

The method calls are specifically defined and consistent across all function blocks that implement the same interface. In a function block, the methods have to be filled with implementation code.

☞

- An interface is a collection of method prototypes (interface methods/interface properties).

  Declaring variables within an interface is not permitted.

  The method prototypes of the interface may only define input, output and "in/output" variables. An interface and its method prototypes has neither an implementation part nor actions.
- Variables that are declared with the type of an interface are always treated as references.
- The methods/properties assigned to a function block that implements an interface has to be named exactly as in the interface and has to contain exactly the same variables.

Adding an interface: To add the "Interface" object to the project, highlight the "Application" node and select **Add ▸ Interface...** in the context menu.

To provide the object project-wide, add it to the "General module" folder.

Alternatively, use the mouse to drag the "Interface" object from the "PLC Objects" library to the desired position.

Enter a name (<InterfaceName>) into the "Add Object" dialog of the interface.

The "Extends:" option can also be selected if the interface is supposed to extend, page 36, another one, which means that its methods definitions apply automatically in addition to those defined locally.

If the interface "Interface1" extends the interface "Interface_base", all methods/properties described by "Interface_base" are also available in "Interface1":

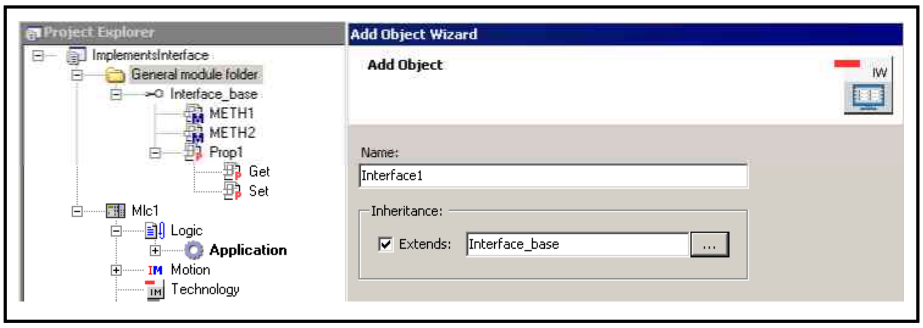


Fig.2-8: Example for extending an interface

After the settings have been confirmed with "Finish", the "Interface" object is created in the Project Explorer. Start programming by double-clicking the "Interface" object or via **Open** in the context menu to open the editor.

Declaration:

```
INTERFACE <interface name>
```

Example for an interface that extends another one:

```
INTERFACE <interface name B> EXTENDS <interface name A>
```

Example:

```
INTERFACE Interface1 EXTENDS Interface_base
```

Adding methods/properties To complete the definition of the interface, the desired methods/properties have to be added. To do this, the interface object is selected in the Project Explorer and opened via **Add** in the context menu in the "Add Object" dialog. Enter the name of the return type into the "Add Object" dialog of the interface.

Use the [...] button to open the "input assistance", page 98, to select the return type.

Add all desired methods/properties and note that these here only contain the declarations of input, output and input/output variables, but no implementations.

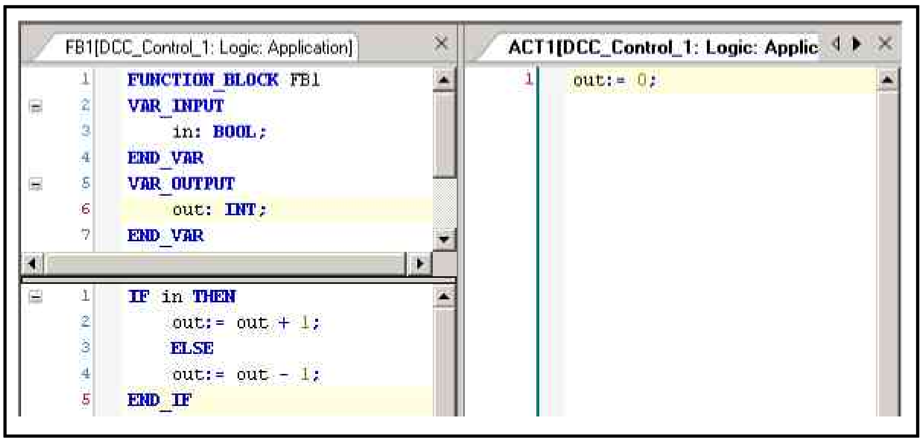
2.6.9 Action (ACTION)

Actions can be assigned to function blocks, page 33, and programs, page 29,.

An action contains further implementation code that can be written in a different language than the "basic" implementation.

Each action receives a name.

An action has no individual declarations. It operates with the data of the function block or the program to which it is assigned. It uses its input, output and local variables.


FB1[DCC_Control_1: Logic: Application]
ACT1[DCC_Control_1: Logic: Applic

```
FUNCTION_BLOCK FB1
VAR_INPUT
    in: BOOL;
END_VAR
VAR_OUTPUT
    out: INT;
END_VAR
```

```
IF in THEN
    out:= out + 1;
    ELSE
    out:= out - 1;
END_IF
```

```
out:= 0;
```

Fig.2-9: Example of a function block action

In this example, each call of the function block "FB1" increases the output variable "out" based on the value of the input variable "in".

Calling the function block action "ACT1" resets the output variable "out" zero.

In both cases, the same variable "out" is written, i.e. the variable declaration for "FB1" also applies to its action "ACT1".

Adding an action: To add an action, highlight a function block or a program object and select **Add ▸ Action** in the context menu. Alternatively, use the mouse to drag the "action" object from the "PLC Objects" library to the desired position.

In the "Add Object" dialog, enter a name for the action and select the implementation language (programming language).

Action call: Syntax:

`<program name>.<action name>`

or

`<FB instance name>.<action name>`

If an action is to be called in the function block to which it is assigned, specifying the action name is sufficient.

Examples for calling an action from another POU: *Declaration for all examples:*

```
PROGRAM Plc_main
VAR
```

```
    Inst: Counter;
END_VAR
```

Calling the "Reset" action in another function block programmed in "IL" (instruction list):

Example in IL:

```
CAL Inst.Reset(In := FALSE)
LD Inst.out
ST ERG
```

Calling the "Reset" action in another function block programmed in "ST" (structured text):

Example in ST:

```
Inst.Reset(In:= FALSE);
Erg:= Inst.out;
```

Calling the "Reset" action in another function block programmed in "FBD" (function block diagram):

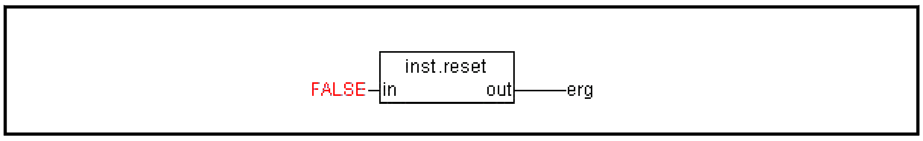


Fig.2-10: Calling the action from another function block

2.6.10 External POUs (Functions, Function Blocks, Methods)

No code is generated by the programming system for external functions, function blocks or methods.

To create an external function block, carry out the following steps:

1. Add a POU to the global "General module" folder. To do so, highlight the "General module" folder and select **Add** ▸ **POU** in the context menu. Use the "Finish" button to confirm your entries.

   Use **Open** in the context menu to open the editor and define the corresponding input and output variables.

☞ Local variables have to be declared in "external" function blocks that may not be declared in external functions or methods!

Likewise, note that "VAR_STAT" variables cannot be used in the runtime system!

2. Define the POU as "external" POU:

   To do this, highlight the POU object in the Project Explorer and open the "Properties" dialog in the context menu via **Properties**.

   There, open the "Build" tab and enable the "External implementation (late linking in the runtime system)" option.

An equivalent function, function block, etc has to be implemented in the runtime system.

At program download, the equivalent function block in the runtime system is browsed through for each "external" POU and integrated if found.

2.6.11 Global Variable List - GVL

Icon: 🌐

A global variable list (GVL) is used to declare global variables, page 518,.

If a GVL is present in the "General module" folder, the variables contained there are available across the entire project.

If a GVL is assigned to a certain application, the variables apply in that application.

To add a GVL, highlight the "General module" folder or the "Application" node and use **Add ▸ Global variable list** in the context menu. Alternatively, use the mouse to drag the "global variable list" object from the "PLC Objects" library to the desired position. Specify a name for the GVL in the "Add Object" dialog.

Double-click on the GVL object or use **Open** in the context menu to work in the GVL editor, page 367,.

If the target system supports the network functionality, the variables of a GVL can become network variables, page 71, and used in data exchanges with other devices in the network. To do this, the corresponding network properties have to be defined for the GVL.

2.6.12 Global Network Variable List - GNVL

Icon:

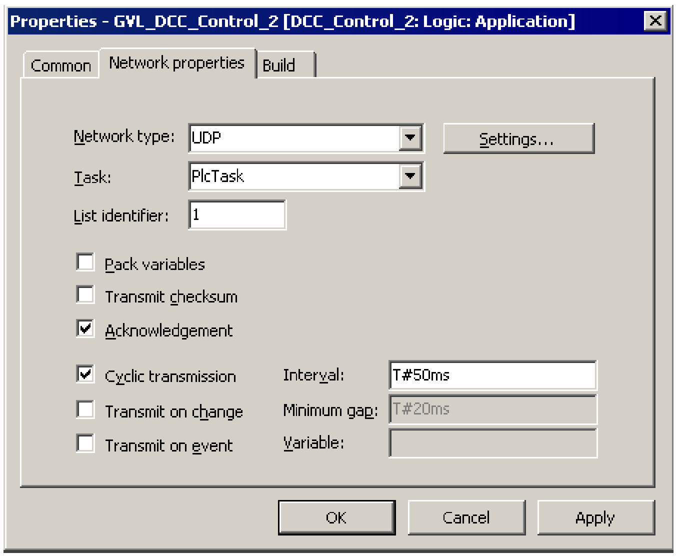
A global network variable list (global NVL, GNVL) includes variables that are defined in another network device as network variables.

☞ The data volume that can be exchanged using a global network variable list is limited (max. 255 bytes).

It is only used below an application in the Project Explorer, page 63,.

☞ A GNVL object can be added to an application if at least one GVL, page 52, with special network properties is present in another device.

Click on **Properties** in the context menu of the GVL and open the "Network properties" tab to assign network properties.


Properties - GVL_DCC_Control_2 [DCC_Control_2: Logic: Application]

Common | Network properties | Build

Network type: UDP
Task: PlcTask
List identifier: 1

☐ Pack variables
☐ Transmit checksum
☑ Acknowledgement
☑ Cyclic transmission — Interval: T#50ms
☐ Transmit on change — Minimum gap: T#20ms
☐ Transmit on event — Variable:

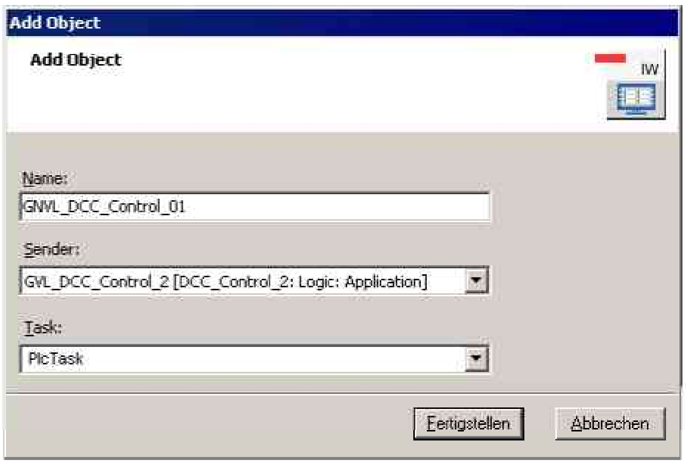
Fig.2-11: "GVL Properties" dialog, network variables

A detailed description of the setting options can be found in "Network variables", page 71.

To add a GNVL, highlight the "Application" node and use the context menu items **Add ▸ Global network variable list** in the context menu. Alternatively, use the mouse to drag the "Global Network Variable List" object from the "PLC Objects" library to the desired position.

If there are several GVLs in the network one can be chosen from the "Sender" selection list, when adding the GNVL in the Add Object dialog, page 234,. A GNVL in one device does not always correspond exactly to a GVL in another device. When creating a GNVL, a task responsible for handling the network variables, page 71, has to be defined as well.

The settings for a global network variable list can always be edited later on in the object properties.


Add Object

Add Object

Name: GNVL_DCC_Control_01

Sender: GVL_DCC_Control_2 [DCC_Control_2: Logic: Application]

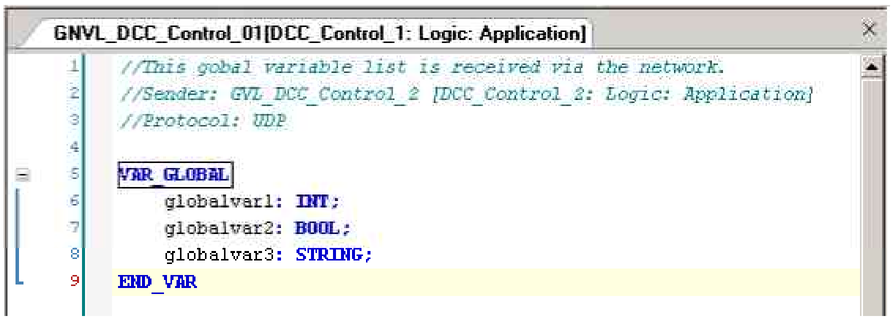
Task: PlcTask

Fig.2-12: "Add Object" dialog, global NVL

A GNVL is displayed in an editor window (NVL editor, page 382), but users cannot edit the content.

The list shows the same variable declarations as the respective GVL. If the GVL is changed, the GNVL is updated respectively.

Above the declarations, a comment is automatically added to the GNVL which contains information on the sender (device path for the device where the GVL is located), the GVL name and the protocol type.


GNVL_DCC_Control_01[DCC_Control_1: Logic: Application]

```
//This gobal variable list is received via the network.
//Sender: GVL_DCC_Control_2 [DCC_Control_2: Logic: Application]
//Protocol: UDP

VAR_GLOBAL
    globalvar1: INT;
    globalvar2: BOOL;
    globalvar3: STRING;
END_VAR
```

Fig.2-13: Example of a global network variable list

For general information on using network variables, see "Network variables", page 71.

2.6.13 Persistent Variables (VAR PERSISTENT)

Icon: 𝕋

Persistent variables are only re-initialized at control reboot or reset. In particular, they maintain their value after a download.

See also "Remanent Variables", page 519.

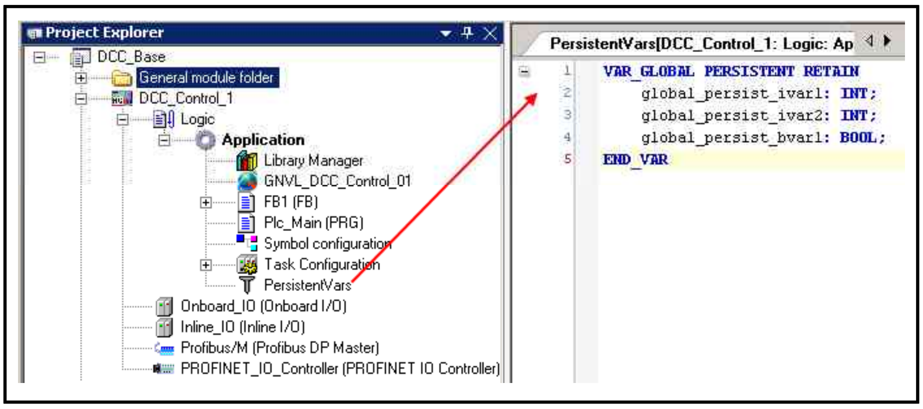
This object is a global variable list, although it only contains the persistent variables of an application. That means that the object has to be assigned to an application.

To add the "PersistentVars" object, highlight the "Application" node and use **Add ▸ PersistentVars** in the context menu. Alternatively, use the mouse to drag the "PersistentVars" object from the "PLC Objects" library to the desired "Application" node.

In the Add Object dialog, page 234,, specify the "PersistentVars" object a name.

Double-click on the "PersistentVars" object or use **Open** in the context menu to work in the editor.

A persistent variable list is created in the GVL editor, page 367,. `VAR_GLOBAL PERSISTENT` is automatically specified in the first line.



```
VAR_GLOBAL PERSISTENT RETAIN
    global_persist_ivar1: INT;
    global_persist_ivar2: INT;
    global_persist_bvar1: BOOL;
END_VAR
```

Fig.2-14: Example of a persistent variable list

Persistent variables are only re-initialized when the control is rebooted or reset.

2.6.14 Text List

Text lists are used to manage texts that can be displayed in the visualization. These can be error messages that output a defined text from the text list when an error occurs for example.

The "Text List" object is assigned and managed globally in the "General module" folder or in an application. It is the basis for

1. **Multilingualism (multilanguage support)** for "static" and "dynamic" texts and tooltips in visualizations and in handling alarms and
2. **Dynamic text change**.

Text lists can be exported and imported, page 60,.

Export is required if a language file, page 628, has to be provided in XML format; see Export and Import Text Lists, page 60.

Text list formats include text format and XML. Support of "Unicode" can be activated, page 59,.

Each text list is uniquely defined using its namespace. It contains character strings that are uniquely referenced within the list by an ID (identifier, index) and a language abbreviation (any respective character string). The text list to use is specified in the configuration of a visualization element.

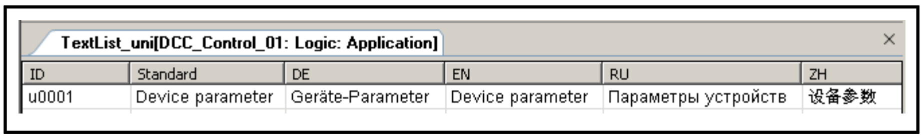
The respective text is then displayed in online mode based on the language just set in the programming system. Each text in the list is at least available in the "standard/default" language. If there is no entry in the text list that matches the language currently set in IndraLogic, the entry defined as default is used. Each text can contain formatting specifications, page 60,.

Structure of a text list:

ID (Index)	Standard	<Language 1>	<Language 2>	.... <Language n>
<unique character string>	<Text abc in the default language>	<Text abc in language 1>	<Text abc in language 2>	...
<unique character string>	<Text xyz in the default language>	<Text xyz in language 1>	<Text xyz in language 2>	...

☞ A "Text Lists" object can include the characters of any language.

With regard to further processing, decide whether the respective characters can be processed as plain text or Unicode.


TextList_uni[DCC_Control_01: Logic: Application]

ID	Standard	DE	EN	RU	ZH
u0001	Device parameter	Geräte-Parameter	Device parameter	Параметры устройств	设备参数

Fig.2-15: Example for a Unicode "Text Lists" object

Text list types

There are two types of text that can be used in visualization elements and there are two types of text lists respectively:

1. **GlobalTextList for static texts:**

   In contrast to dynamic texts, static texts in a visualization cannot be exchanged in online mode using a variable. Only the local country code can be switched as described above.

   A static text is assigned to a visualization element via Property, page 451, "Text" or "Tooltip" of the "Texts" category.

   As soon as the first static text is defined in a project, a text list with the name "GlobalTextList" **is automatically created** as object in the "General module" folder. First, the list contains the defined text in the "Default" column and an automatically assigned integer (beginning with 0) as text "ID". Other static texts are then added as soon as they are defined in the properties of a visualization element. The ID number is then each time incremented by 1.

   If a static text is entered within a visualization element (e.g. if the text "Example" is entered below the "Texts" "Text" square), it is searched for this text in the GlobalTextList.

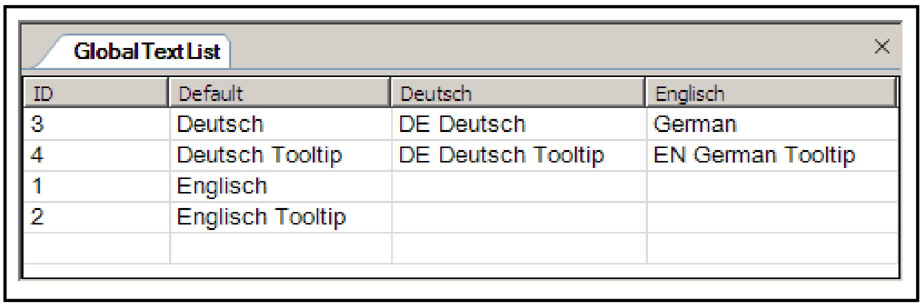
   - If the text is found (e.g. ID "4711", text "Example"), the value 4711 is entered in the element in an internal "TextId" variable. This creates a connection between the element and the line within the GlobalTextList.
   - If the text is not found, a new line is entered in the GlobalTextList (e.g. ID "4712", text "Example"). The value 4712 is applied to the element in the internal "TextId" variable.

This means that for each modification of a static text within the visualization there might also be a modification within the GlobalTextList.

A global text list can also be created explicitly in the visualization editor context menu via **Create Global Text List**.

Alternatively, a text list can also be created in the main menu via **VI Logic Visualization ▸ Create Global Text List**.

"GlobalTextList" is a special text list, in which the identifiers (IDs) for the individual text entries are handled implicitly and users cannot edit them in IndraLogic. This list cannot be deleted. But it can be exported, page 60, edited externally and then re-imported, page 60,. In this case, it is checked during the reimport whether the identifiers still match those specified in the configuration of the respective visualization element. If necessary, an implicit update of the IDs is made in the element configuration.



Global Text List

ID	Default	Deutsch	Englisch
3	Deutsch	DE Deutsch	German
4	Deutsch Tooltip	DE Deutsch Tooltip	EN German Tooltip
1	Englisch		
2	Englisch Tooltip		

Fig.2-16: Example of a GlobalTextList

2. **Text list for dynamic texts:**

Dynamic texts can be exchanged dynamically in online mode (see above). The text index (ID), a character string, has to be unique within the text list used and, in contrast to a "GlobalTextList" has to be specified by the user. Another difference from "GlobalTextList" is that text lists for dynamic texts have to be created explicitly. To add a text list, highlight the "Application" node and select **Add ▸ Text List** from the context menu. Alternatively, use the mouse to drag the "text list" object from the "PLC Objects" library to the "Application" node.

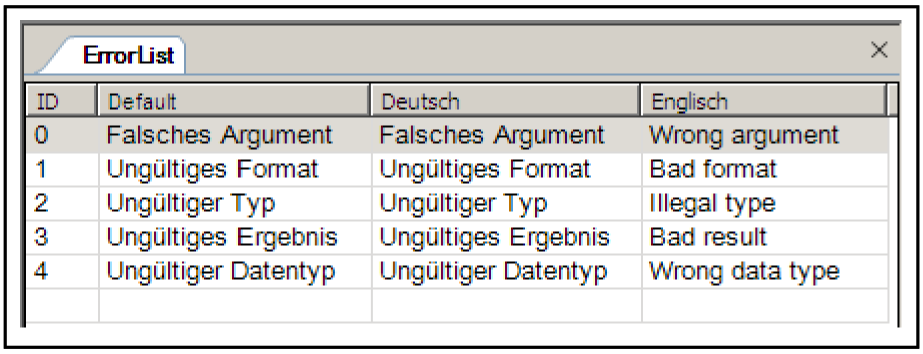
In the Add Object dialog, page 234,, specify the text list a name and confirm with "Finish".

All dynamic text lists available in the project are provided when configuring the property, page 451 , "Dynamic Texts" / "Text List" of a visualization element.

To enter the name of a text list and a text index (ID) from the list, the corresponding text is displayed in online mode. If the ID is not entered as absolute, but using a project variable instead, the text can be switched dynamically using this variable.

In contrast to "GlobalTextList", the IDs are not automatically checked and updated when dynamic text lists are re-imported!

Ensure that the index IDs are not changed when the exported lists are edited!



ErrorList

ID	Default	Deutsch	Englisch
0	Falsches Argument	Falsches Argument	Wrong argument
1	Ungültiges Format	Ungültiges Format	Bad format
2	Ungültiger Typ	Ungültiger Typ	Illegal type
3	Ungültiges Ergebnis	Ungültiges Ergebnis	Bad result
4	Ungültiger Datentyp	Ungültiger Datentyp	Wrong data type

Fig.2-17: Example of a dynamic text list called "ErrorList"

The following is a description of an example.

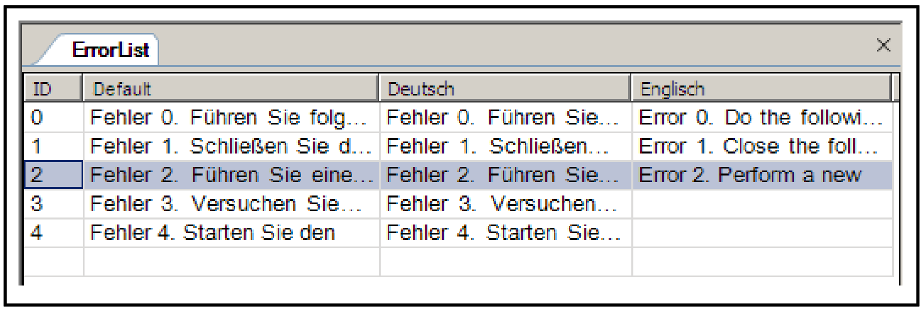
Example:

Dynamic text list

Configure a visualization element that is supposed to output the corresponding error messages when an error occurs. The application processes errors that are identified via numerical IDs - assigned to an integer variable "ivar_err".

Proceed with the following steps:

1. Provide a dynamic text list called "ErrorList" where the error message texts for the error IDs "0" to "4" are defined in "German", "English" and "Default language". See the following figure.



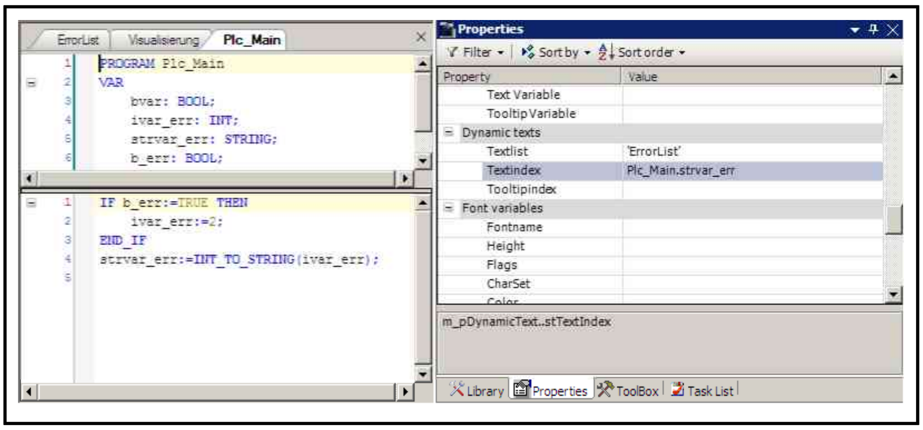
ErrorList

ID	Default	Deutsch	Englisch
0	Fehler 0. Führen Sie folg...	Fehler 0. Führen Sie...	Error 0. Do the followi...
1	Fehler 1. Schließen Sie d...	Fehler 1. Schließen...	Error 1. Close the foll...
2	Fehler 2. Führen Sie eine...	Fehler 2. Führen Sie...	Error 2. Perform a new
3	Fehler 3. Versuchen Sie...	Fehler 3. Versuchen...	
4	Fehler 4. Starten Sie den	Fehler 4. Starten Sie...	

Fig.2-18: "ErrorList" example

2. Declare a STRING variable, e.g. `strvar_err` to use error IDs in a visualization configuration.
3. To assign the value of `ivar_err` to the variable `strvar_err`, use `strvar_err:=INT_TO_STRING(ivar_err);`

Now, "strvar_err" can be used as a text index parameter in the configuration of the "dynamic texts" properties of a visualization element. The element displays the corresponding error message in online mode.



```
PROGRAM Plc_Main
VAR
    bvar: BOOL;
    ivar_err: INT;
    strvar_err: STRING;
    b_err: BOOL;
```

```
IF b_err:=TRUE THEN
    ivar_err:=2;
END_IF
strvar_err:=INT_TO_STRING(ivar_err);
```

Property	Value
Text Variable	
Tooltip Variable	
Dynamic texts	
Textlist	'ErrorList'
Textindex	Plc_Main.strvar_err
Tooltipindex	
Font variables	
Fontname	
Height	
Flags	
CharSet	
Color	

m_pDynamicText..stTextIndex

Fig.2-19: Example: Project variables processing the error ID. Configuration of a visualization element ("Properties" dialog) to output the error messages.

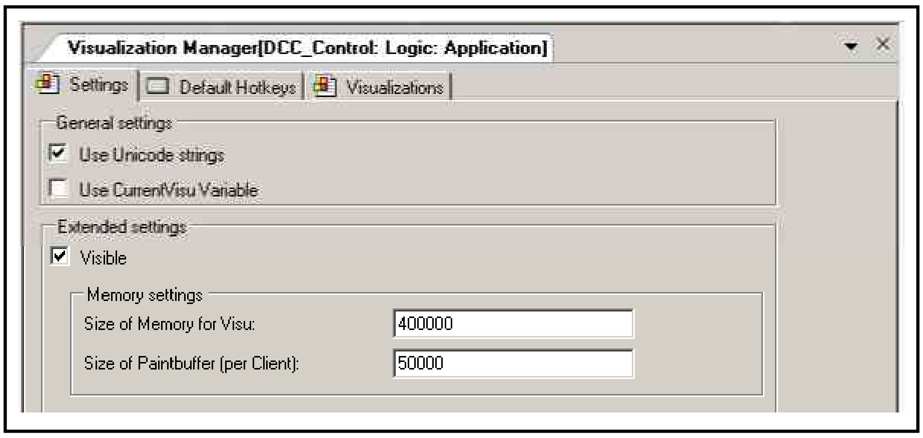
Creating a text list

- If the "Text List" object is to be assigned to an application, highlight the "Application" node in the Project Explorer. If the "Text List" object is to be available project-wide, add it to the "General module" folder. To do this, highlight the "Application" node or the "General module" folder and select **Add ▸ Text List** in the context menu. Alternatively, use the mouse to drag the "Text List" object from the "PLC Objects" library to the respective position.
- To create the text list "GlobalTextList" for static texts, enter any text into the "Properties" category "Texts" at the property "Text" when configuring a visualization element. This procedure automatically generates the list. Alternatively, generate a text list for static texts via **Create global text list** in the context menu. The "Create global text list" command is also available via **VI Logic Visualization** in the main menu.
- To open an existing text list for editing, use **Open** in the context menu or double-click on the object entry. See also Structure of a Text List, page 56 to see how a text list is structured.
- To edit a field in the text list, proceed as follows:
  1. Click on the field to select it.
  2. Click it again or press the <space bar> to open an input field.
  3. Enter any character string and close the field with <Enter>.
  4. Use the arrow keys to move to the next or previous field.

Support for Unicode format

To support the Unicode format, proceed as follows:

1. Open the visualization manager editor window (double-click the entry in the Project Explorer or use the context menu).
2. Enable the "Use Unicode strings" option.


Visualization Manager[DCC_Control: Logic: Application]

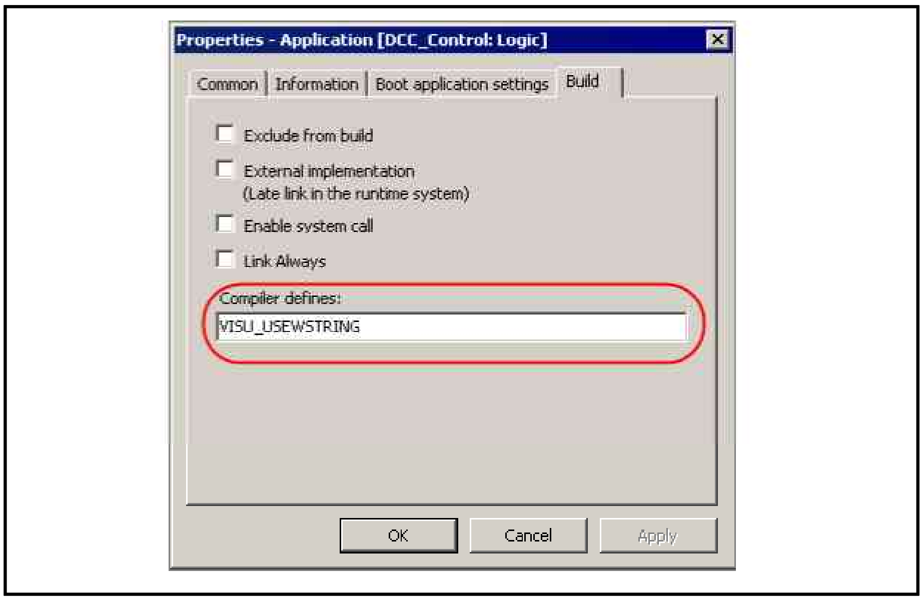
General settings
- [✓] Use Unicode strings
- [ ] Use CurrentVisu Variable

Extended settings
- [✓] Visible

Memory settings
Size of Memory for Visu: 400000
Size of Paintbuffer (per Client): 50000

Fig.2-20: "Visualization Manager" editor window

3. Use "OK" to confirm your entries.
4. Enter a special compiler instruction for the application. To do this, highlight the application in the Project Explorer and select **Properties** in the context menu. Click on the "Build" tab and - in the "Compiler defines:" field - enter "VISU_USEWSTRING":


Properties - Application [DCC_Control: Logic]

- [ ] Exclude from build
- [ ] External implementation (Late link in the runtime system)
- [ ] Enable system call
- [ ] Link Always

Compiler defines:
VISU_USEWSTRING

Fig.2-21: "Application" properties, entering compiler definition

5. Use "OK" to confirm your entries.

Exporting and importing text lists

Static and dynamic text lists can be exported in text or XML format.

Exported files can also be used to add external texts, e.g. from a compiler. Note that only files in text format (*.txt, *.csv) can be imported again.

Detailed descriptions of the corresponding commands can be found in:

Menu items:

- Import/Export text lists, page 288

Formatting texts

The texts can contain formatting specifications (%s,%d,...) that enable to return current variable values in the text for example. The allowable formatting definitions can be found in "Visualization," page 628,

The text definition is evaluated in the following sequence in order to display the respective current text:

1. The text to be evaluated is determined by list name and text ID.
2. If the text contains formatting specifications, they are replaced by the value of the corresponding variable.

Subsequent delivery of compilations

Adding the "GlobalTextList.csv" (subsequent file) to the directory used for loading text lists allows compilations to be delivered subsequently. When the boot project is started, it is determined if a subsequent file exists and the compilations are compared with the text list files. Both new and modified compilations are applied to the text list files.

Afterwards, the "GlobalTextList.csv" file is marked as loaded. This way, subsequent delivery of texts only affects the start-up time for the boot project once.

Intellisense for text input

A text template file can be specified using the visualization options. All texts in the "Default" column of this file are included in a list that is used for "TextIntellisense". A file that was previously generated with the export command can be used as a text template file.

Multiple user mode

By using source code management in IndraLogic 2G, it is possible for several users to work on a project simultaneously. Note the following points:

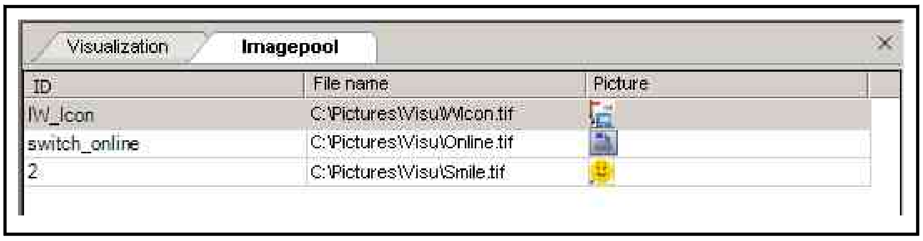
- If a static text is modified within a visualization element, the visualization has to have write access and perhaps the GlobalTextList as well (see GlobalTextList). If the GlobalTextList does not have write access, none of the texts in the visualizations should be modified. But if they are modified, the text IDs might no longer match the texts in a visualization element.
- The Check Visualization Text IDs, page 293, command can determine these kinds of errors in all visualizations.
- The Update Visualization Text IDs, page 293, command can automatically correct these error cases. To do this, all visualizations with error cases and the GlobalTextList have to have write access.

A delivery that with error cases can lead to the wrong texts appearing in the visualization if the language is switched. If no cases of error are reported for a project, the language file can be compiled and delivered subsequently.

2.6.15 Image Pool

Image pools are tables that define the path, a preview and an identifier (ID, character string) for each image file. By entering the ID and - for unique addressing - the name of the image pool, an image file can be referenced if it is used in a visualization in the project, for example (in the configuration of the properties of a visualization element, see Using Image Files from Image Pools, page 62).

Structure of an image pool:



ID	File name	Picture
IW_Icon	C:\Pictures\Visu\Icon.tif	
switch_online	C:\Pictures\Visu\Online.tif	
2	C:\Pictures\Visu\Smile.tif	

Fig.2-22: Example of an image pool

"ID":

Identifiers as character string, e.g. "IW_Icon", "switch_online", "2"); unique referencing of an image file is achieved by combining the name of the image pool with the image file ID (e.g. "List1.basic_logo").

"File name":

Image file path (e.g. "C:\Pictures\Visu\Online.tif")

"Image":

Image preview

Creating and editing an image pool

One project can include several image pools.

If the project does not yet include an image pool, as soon as the first "image" element is added to the visualization and an ID (static ID) is entered in the element properties, an image pool is automatically created with the default name "GlobalImagePool" along with an entry for the selected image file. "GlobalImagePool" is a global pool that is always searched first when an image file is is to be used. Individually named pools can also be used.

Image pools can be created manually as follows:

- via the main menu using **VI Logic Visualization** ▸ **Generate Global Image Pool**
- via the context menu via **Generate Global Image Pool** if the mouse is in the visualization editor
- via the context menu below an application or the "General module" folder via **Add** ▸ **Image Pools...**.
- with the mouse by dragging and dropping from the "PLC Objects" library.

To add an image file manually to an image pool, place the focus in the **ID** field of the first empty line in the pool table, press the <space> bar to open an input field and enter any character string as ID. If the ID entered is already used in the table, a numerical digit is automatically added, beginning with 0 and incremented by 1 each time the ID is copied. Then place the cursor in the "File Name" field. Here, use the <space> bar and the ... key to open the "Image Selection" dialog to enter the path of the desired image file.

Using image files from image pools

Note the following if the ID of an image file is present in several image pools:

- Search order:

  When an image is selected that is managed in the "GlobalImagePool", the name of the image pool does not have to be specified. The search order for image files corresponds to that for global variables:

  1. "GlobalImagePool" in the "General module" folder
  2. Image pools that are assigned to currently active applications
  3. Image pools that, in addition to "GlobalImagePool", are located in the "General module" folder
  4. Image pools in libraries

- Unique access:

  Address the desired image file directly and uniquely by using the ID of the name of the image pool as prefix.

  Syntax:

  `<name of image pool>.<image id>` (e.g. for the example shown in the figure above: "imagepool.IWIcon").

1. Using an image file in a visualization element, page 446, of the type "Image":

   If an "Image" element is added to a visualization, either a static or dynamic element can be specified, where the dynamic element can be exchanged based on a variable in online mode:

- Static images:

  Enter the image file ID or the name of the image pool + ID in the configuration of the element ("Static ID" property). Note here the information on Search Order and Unique Access, page 62 (see above).

- Dynamic images:

  Enter the variable that defines the image file, e.g. "Plc_Main.imagevar" into the configuration of the element ("Bitmap ID variable" property).

2. Using an image file for the visualization background, page 304:

   An image can be entered in the background definition, page 304, for a visualization. As described above for a visualization element, the image file can be entered using the name of the image pool and the file name.

2.6.16 Visualization

Information on the visualization in IndraLogic 2G and on the visualization editor can be found in Visualization, page 625,.

2.6.17 POUs for Implicit Checks

These special POUs can be added to an application to equip it with available implicit monitoring functionalities. At runtime, they check for array limits or subsection types, the validity of pointer addresses or division by 0.

In the "Add Object" dialog in the "POUs for Implicit Checks" category, the following functions are available:

- CheckBounds, page 560
- CheckDivInt, page 571
- CheckDivLInt, page 571
- CheckDivReal, page 571
- CheckDivLreal, page 571
- CheckRange, page 566
- CheckRangeUnsigned, page 566
- CheckPointer, page 557

After adding a POU for monitoring purposes, it opens in the editor that corresponds to the selected implementation language. A suggestion for the implementation is made in the ST editor and can be adapted as desired.

To prevent multiple linking, a monitoring function that has already been added can no longer be selected in the "Add Object" dialog. If all types of monitoring functions have already been added, the entire "POUs for Implicit Checks" category is removed from the dialog.

To maintain the functionality of monitoring functions, the declaration part may not be modified.

Also refer to

- Floating point Exceptions in the PLC Program, page 335

2.7 Devices in the Project Explorer

All objects required for executing an application, page 66, (a control program) on a device (control, PLC) are managed in the Project Explorer in a tree structure.

These objects are also referred to as "Resource" objects. Device objects, application objects, task configuration and task are "Resource" objects.

Programming objects such as individual POUs, global variable lists and library managers can be managed below a control in the Project Explorer and can then be used only for your application.

☞ Globally applicable programming objects are managed in the "General module" folder in the Project Explorer.

To convert device references when opening projects created in another format, see

Data Transfer, page 115.

General information on the device object tree

- The root node in the Project Explorer is the name of the project given when a new project is created. <project name>.
- The configuration trees for the "control configuration " and "task configuration" that were handled in separate windows in IndraLogic 1.x are integrated into the device object tree in IndraLogic 2G. The configuration of the devices and task parameters is carried out in separate editor dialogs.

  See Task Configuration, page 67, Control Configuration, page 68.

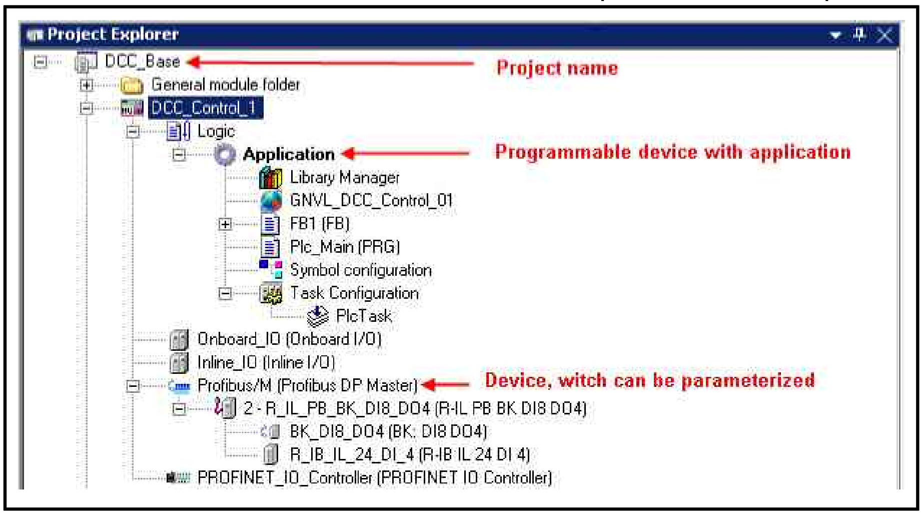
  This way, the structure of the hardware environment to be controlled is illustrated in the device object tree with the corresponding arrangement of objects and it is possible to superimpose a heterogeneous system of controls with multiple networking and underlying field buses.

See the rule for arranging objects below the device node in the following.

- A "Devices" object represents a certain hardware (target system).

  Examples: control device, drive, I/O module, monitor.
- An entry in the Project Explorer shows the icon, the symbolic device name (which can be edited in the tree) and the device type behind it (= device name as defined in the device description).
- There are "programmable" devices and devices that can be "parameterized". The device type determines the possible insertion position below the device node and the selection of objects that can be added below the device.

  Programmable devices automatically obtain an additional "Logic" node below the device object. The objects required for programming the device (visualizations, GVLs, text lists, etc.) can then be added below this node:



```
Project Explorer
DCC_Base                                   <- Project name
  General module folder
  DCC_Control_1
    Logic
      Application                          <- Programmable device with application
        Library Manager
        GNVL_DCC_Control_01
        FB1 (FB)
        Plc_Main (PRG)
        Symbol configuration
        Task Configuration
          PlcTask
    Onboard_IO (Onboard I/O)
    Inline_IO (Inline I/O)
    Profibus/M (Profibus DP Master)        <- Device, witch can be parameterized
      2 - R_IL_PB_BK_DI8_DO4 (R-IL PB BK DI8 DO4)
        BK_DI8_DO4 (BK: DI8 DO4)
        R_IB_IL_24_DI_4 (R-IB IL 24 DI 4)
    PROFINET_IO_Controller (PROFINET IO Controller)
```

Fig.2-23: Devices in the Project Explorer

- In a project, page 26,, one or more programmable devices can be configured irrespective of the manufacturer or type.
- The configuration of a device with respect to communication, parameters or I/O mapping is carried out in the "Device Editor dialog" that can be opened by double-clicking on the device entry (a detailed description can be found in Device Editor, page 331,).
- In "online mode", an icon in front of a device entry indicates whether the device is currently connected or not connected. Additional diagnostic information can be found in the respective logbook, page 332, in the "Status" category.

See the following notes and rules for arranging, page 65, objects in the device.

Arranging and configuring objects in the device:

- The object types that can be added depend on the currently selected position in the tree.

  Example:

  Modules for a DP PROFIBUS slave cannot be inserted without inserting the respective slave object before.

  Applications cannot be added below non-programmable devices.

  Moreover, the only devices that can be selected for insertion are those that have been correctly installed in the local system and are suitable for use in the currently selected position in the tree.

  To add an object, highlight the position in the tree where the object is to be inserted and use **Add** ▸ **<ObjectType>** in the context menu. Alternatively, insert objects using the mouse to drag them from the library to the corresponding position.

☞ Programmable devices can only be added from the "Drive and Control" library. Highlight the desired device (e.g. IndraLogic XLC L65) and use the mouse to drag it to the root node.

- The arrangement of objects below an application is sorted alphabetically and by object type. It is not possible to change to any position. On the other hand, the objects below (e.g. actions, transitions) can be positioned as desired by using the mouse to drag them to the corresponding position.

- A device already added to the Project Explorer can be replaced by another version of the same device or by a device of another type. The configuration tree indented below the device can be retained as much as possible. To do this, highlight the device node and select **Update Device** in the context menu.
- Devices can be installed and uninstalled in the Device Database... dialog. The installation is based on "device description files" in XML format. The default extension for a valid description file is ***.devdesc.xml**. However, bus-specific description files, e.g. *.gsd files (PROFIBUS) can also be installed using the "Device Repository" dialog.
- A device is added as node in the Project Explorer. If these are defined in the device description, the subnodes are automatically added as well. In turn, a subnode can also be a programmable device.
- Further devices that are installed in the local system and are provided in the library can be added below a "Device" object. For each level, the programmable devices are arranged first (PLC logic), then the rest of the types.
- An application, page 66, is automatically added below the "Logic" node (symbolic node for programmable devices). Only one application is permitted for each device. Then, the other objects required for programming, e.g. data types (DUT), global variable lists (GVL), visualizations, etc. can be attached below an application. A task configuration is automatically added below every application and the corresponding program calls are defined there.

2.8 Application

Icon: ⚙

- An "application" includes several objects required for executing a certain "instance of the control program" on a certain device, page 63, (control, PLC). To do this, the global objects that are managed in the "General module" folder can be instantiated and assigned to a device. This corresponds to the object-oriented programming.

  However, POUs that are purely application-specific can also be used by the application.
- An application is represented by the "Application" object in the device, page 63, (i.e. programmable device) that is automatically added below a Logic node, page 63, with the programmable device. The objects that make up the "resource set" of the application are automatically added below an application.
- An important object for each application is the task configuration, page 67, to check the execution of a program (POU instances or application-specific POUs).

  In addition, objects such as global variable lists, libraries, etc. can be directly assigned to the application, which, in contrast to the objects managed in the "General module" folder, can only be used by this application and its "child" objects; see Arranging and Configuring Objects in a Device, page 65, for the rules.

☞ Note that several applications cannot be used with the same device at present.

- Note that the application to work with in online mode has to be set as "active application", page 237,
.

To do this, highlight the application and select **Set as Active Application** in the context menu. The active application is displayed in the Project Explorer in bold; see the following figure.

- When logging in, page 127, with the application on the control, it is checked which applications are currently on the control and if the application parameters on the control match those in the current project.

  Corresponding messages are output and applications can be deleted on the control.

- Note the "Applications" tab for the Device editor, page 331 to see which applications are currently present on a device and to delete these from the target system.

  Applications can also be displayed that are not represented by a separate object in the device; see "Symbol Configuration", page 306.

2.9 Task Configuration

The task configuration defines one or multiple tasks to control and execute the application program on the control.

It is a required "Resources" object for an application , page 66 , and is automatically added below an application.

A task can call an application-assigned program or a program managed globally in the "General module" folder.

A task configuration can be edited in the task editor, page 428, although the available options depend on the target system.

In online mode, the "task editor" provides a monitoring view with information on cycles, cycle times and task status.

Profiling (monitoring)

☞ The availability of the "profiling (monitoring)" functionality depends on the device type.

Important notes for multitasking systems:

True, pre-emptive multitasking realized on some systems. In this case, observe the following:

As in IndraLogic V1. x, all tasks share a process image.

Reason:

An individual process image for each task would lower performance. The process image can only be consistent with one task. For this reason, when creating a project users are responsible for making sure that in case of conflicts, the input data is copied to a secure area; the same applies to outputs. Possibilities for solving consistency and synchronization problems are provided by the function blocks in the "SysSem" library for example.

Consistency problems can also occur in multitasking systems when other global objects (global variables, function blocks, field buses) are accessed if the objects exceed the data width of the processor (structures or arrays that form a logical unit). A solution is available in the function blocks in the "SysSem" library.

2.10 Communication

2.10.1 Communication, General Information

This section includes information on the following subjects:

- Configuration of a control, page 68
- Data server, page 71

- Network variables, page 71

2.10.2 Control Configuration

The "control configuration" illustrates the target hardware in the programming system in order to make the inputs and outputs and the control parameters and the applications field bus devices accessible. In addition, it enables the available device parameters to be displayed.

The "control configuration" tree that is handled in its own editor in IndraLogic V1. x, is integrated below the "Devices" node in the Project Explorer in IndraLogic 2G where the other objects necessary for running an application on a target system are also arranged. The map of the current hardware configuration below the "Devices" node is simplified in the default device editors by a scan functionality. Information on the device can be found in "Devices in the Project Explorer", page 63,.

The control inputs and outputs for project variables are assigned either with the "AT Declaration", page 512, in the declaration editor or in the "I/O Mapping" dialog of the respective field bus which provides dialogs to configure a device. If a new control is added to the Project Explorer, a preset enters the application automatically added as "mapping application".

2.10.3 Communication in the Control Link via Gateway

The PLC communication from an engineering PC to one or multiple control devices (IndraLogic XLC, IndraMotion MLC or IndraMotion MTX) is always established via an IndraLogic gateway.

the following figure shows the communication relation.

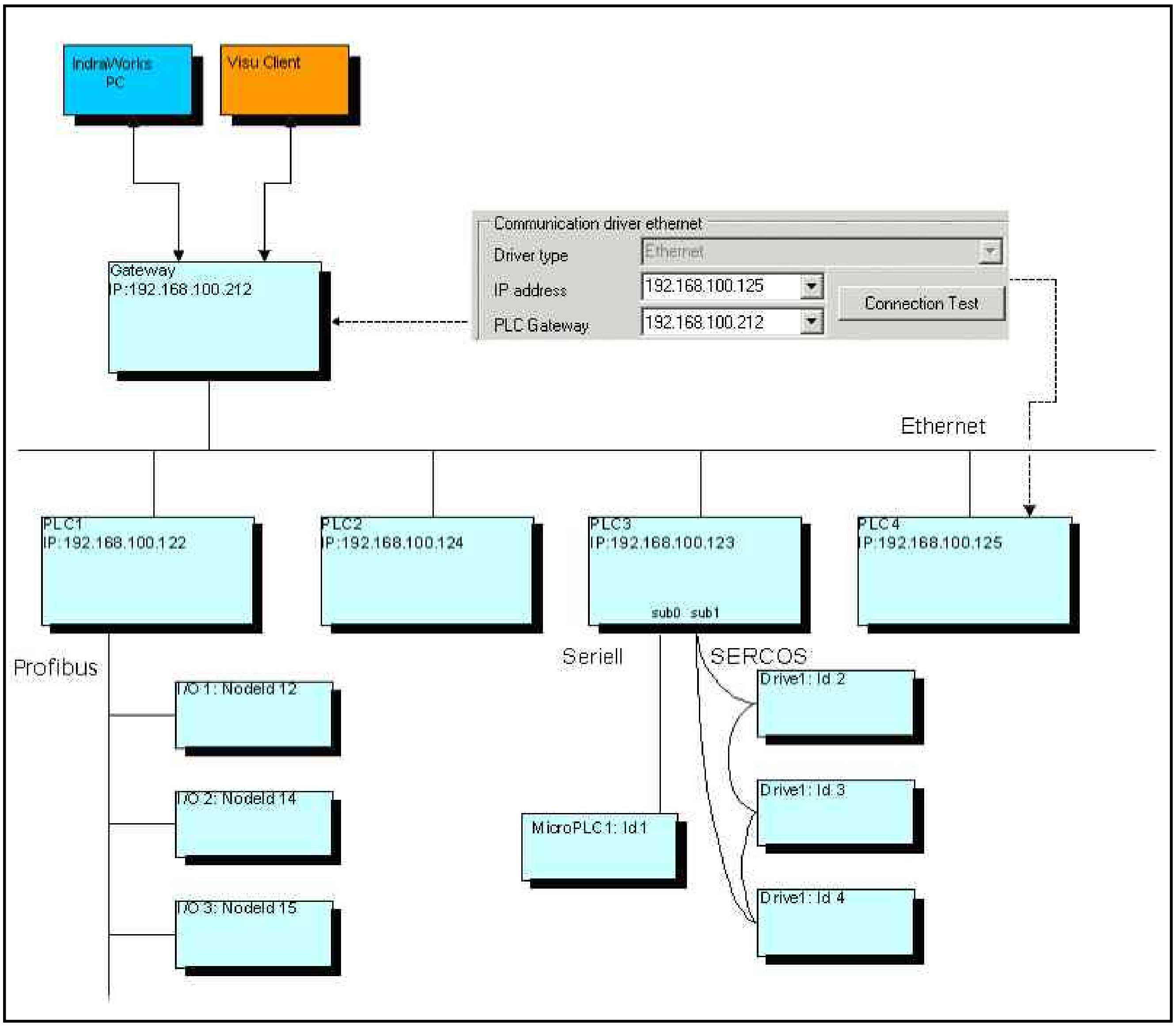


Fig.2-24: Communication relations

The required settings take place in the respective device wizard.

An IndraLogic gateway can either run on the own engineering PC (setting: localhost or 127.0.0.1) or somewhere in the control networks (setting: IP address of the gateway PC).

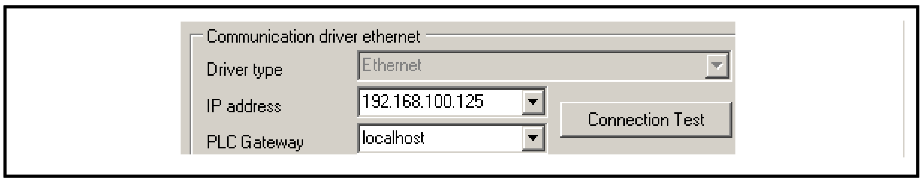

Communication driver ethernet
Driver type: Ethernet
IP address: 192.168.100.125
PLC Gateway: localhost
Connection Test

Fig.2-25: Setting IP and Gateway address

After a successful connection test, the IP address to the control is displayed (here 192.168.100.125). The TCP address can be directly set in the visualization device and in the OPC server.

If the PLC communication is not successful, investigate the following error causes. The error recovery is exemplarily shown under Windows XP. The solution can be different for other Windows versions.

A PLC communication to the device can only be established via the device-engineering interface!

Checking connection to gateway

If there is no gateway connection, the following message is output after a connection test:

- IndraLogic gateway not found.
- IndraLogic gateway: No communication. Gateway offline?

Ensure that the gateway server is running.

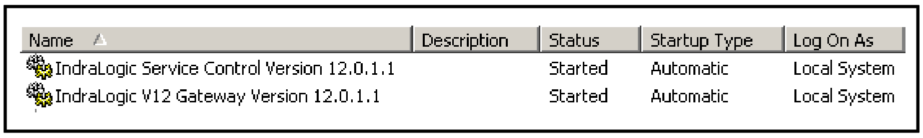
The gateway server is automatically started as service at system start.

Check whether the icon appears in the toolbar at the lower margin of the screen.

If the symbol appears as follows: , the gateway is stopped.

The program icon provides start and stop commands in a context menu opened via the right mouse button. The service can now be started and stopped at any time.

Ensure as well that the services <IndraLogic Service Control> and <IndraLogic V12 Gateway> run in the Windows **Control Panel** under **Administrative Tools** - **Services**.



Name	Description	Status	Startup Type	Log On As
IndraLogic Service Control Version 12.0.1.1		Started	Automatic	Local System
IndraLogic V12 Gateway Version 12.0.1.1		Started	Automatic	Local System

Fig.2-26: List of services (excerpt)

Only one <IndraLogic Service Control> service and <IndraLogic V12 Gateway> service may run at a time.

Close possible further services and set the AutoStart type of these services to "Disabled".

If the connection test keeps on failing, change the settings for the IndraLogic gateway from <localhost> to <127.0.0.1> and subsequently set your own IP address.

Alternatively, the gateway of another engineering PC can be used by entering the IP address below the IndraLogic gateway in the device wizard.

Checking connection to device

If there is no device connection, the following messages are output after a connection test:

- No connection to the device. Device offline?
- No connection to the device. Device offline? Error: <>

Check the control first. Did the control crash? Reboot the control.

Is a firewall enabled? Especially if the control is running on a PC. Disable the firewall.

Were changes made at the Gateway.cfg or GWClient.cfg file? Use the original files without modifications.

Was CoDeSys installed parallelly to IndraWorks? Close the iCoDeSys simulation.

Incorrect response from the device

If the communication with the device is possible at all, device type or device version might not match. In this case, the following messages are output at a connection test:

Different device types:

- The selected target system does not match the connected device.

  ID mismatch:

  requested=1001 0003, online=1001 0103

You are on a third-party device type, e.g. in the device wizard of an MLC L65 device. But the device is an MTX, XLC or MLC with another hardware design (L25, L45).

Different device versions:

- The selected target system does not match the connected device.

  Version mismatching:

  requested=12.6.0.0, online=12.5.2.0

Update the device firmware via the IndraWorks firmware management.

2.10.4 Data Server

A "data server" can be added to an application in order to use remote data sources. In this context, "remote data sources" means that variables ("data items") defined and used in other devices or in the local application can be used.

In contrast to data exchange across network variables (broadcasting), the data server establishes point-to-point connections. Depending on the access flag of the data connections to be exchanged, they are updated in the data source and in the current application each time the respective value changes on the other side.

Using a data server is a faster alternative to provide data via a symbol configuration.

At present, data provided by an OPC server cannot yet be accessed using the data server. In this case, implementing a symbol configuration is still the suitable procedure.

For a description of how a data server is set up and how remote data sources can be used, see Data source editor, page 318,.

2.10.5 Network Variables

Network Variables; Data Exchange between IndraLogic 2G Controls

Network variables have to be defined in fixed variable lists in both the sender and the receiver. Their values are sent via "broadcasting".

Note that this differs fundamentally from the data exchange with a data server, page 71, which uses defined point-to-point connections between the local application and remote data sources.

Based on network variables, 1.x and 2G controls can communicate with each other.

Network variables are handled in...

- Global Variable Lists, page 52, (GVL) in the transmitting device (sender) and in one or more
- Global Network Variable List(s), page 53, (GNVL) in one or more receiving device(s) (receivers)

  GNVLs are displayed in the"network variable list editor", page382,.

The "GVL" and "GNVL" objects that belong to each other have to contain the same variable declarations.

A GVL that is supposed to define network variables has to have special network properties, page 246,.

These are protocol and transfer parameters according to which the variable values are set within the network and can be received by all devices with a matching GNVL.

Note that the transfer of network variables is always in one direction: from the sender (GVL) to the receiver (GNVL)!

However, each device can act as sender or receiver, since each device can handle GVL and GNVL objects.

A prerequisite to exchange network variables is that the suitable "network libraries" are installed. This can be done automatically for the default network functionalities, e.g. for UDP as soon as the network properties for a GVL are set.

The structure of a simple network variable exchange is described in the following example. A GVL is created in the transmitting device and a GNVL in the receiver:

Example:

The following is the preparatory work in a project in which a transmitting device "Dev_Sender" and a receiving device "Dev_Receiver" are created in the Project Explorer:

- Create a POU (program) "prog_sender" below the application in the control Dev_Sender.
- In the task configuration of this application, add the task "Task_S" which calls "prog_sender".
- Create a POU (program) "prog_receiver" below the application in the control "Dev_Receiver".
- In the task configuration of this application, add the task "Task_R" which calls "prog_receiver".

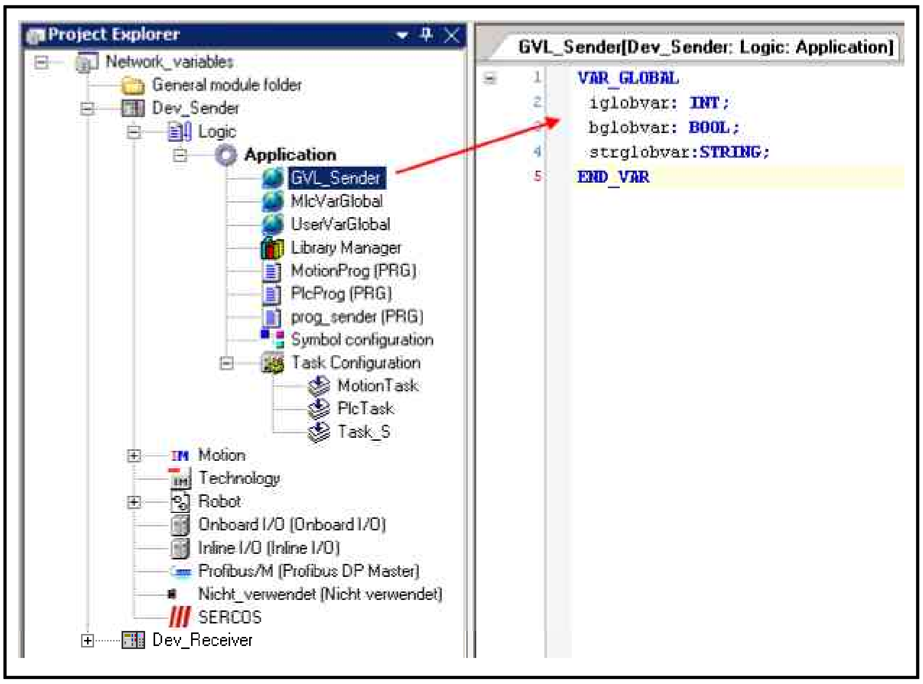
1. Define the global variable list in the transmitting device:

   Highlight the "Application" node in the "Dev_Sender" control. In the context menu, select **Add ▸ Global Variable List** and enter the name "GVL_Sender" in the "Add Object" dialog.

   Use "Finish" to confirm your entries. Double-click on "GVL_Sender" to open the GVL editor and enter the following lines.

GVL_Sender

```
VAR_GLOBAL
 iglobvar: INT;
 bglobvar: BOOL;
 strglobvar:STRING;
END_VAR
```

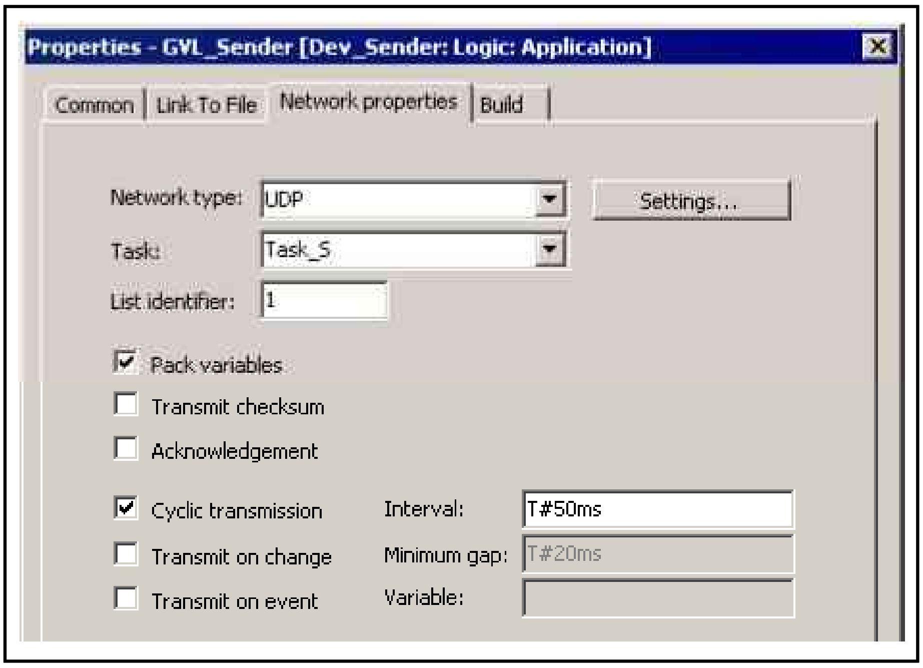


```
VAR_GLOBAL
  iglobvar: INT;
  bglobvar: BOOL;
  strglobvar:STRING;
END_VAR
```

Fig.2-27: Adding a GVL in the transmitting device

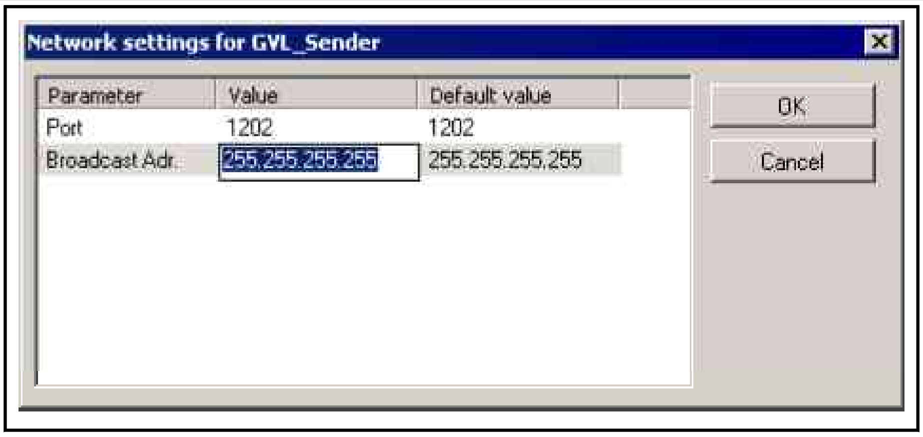
2. Define the network properties of the sender GVL:

   Highlight "GVL_Sender" in the Project Explorer and select **Properties** in the context menu. Open the "Network variables" tab. Set the network properties as follows; see the following figure.



Network type: UDP
Task: Task_S
List identifier: 1
[✓] Pack variables
[ ] Transmit checksum
[ ] Acknowledgement
[✓] Cyclic transmission Interval: T#50ms
[ ] Transmit on change Minimum gap: T#20ms
[ ] Transmit on event Variable:

Fig.2-28: Setting GVL network properties


Network settings for GVL_Sender

Parameter	Value	Default value
Port	1202	1202
Broadcast Adr.	255.255.255.255	255.255.255.255

Fig.2-29: Setting GVL network properties, settings

- Enter the IP address of the receiver control as broadcast address!

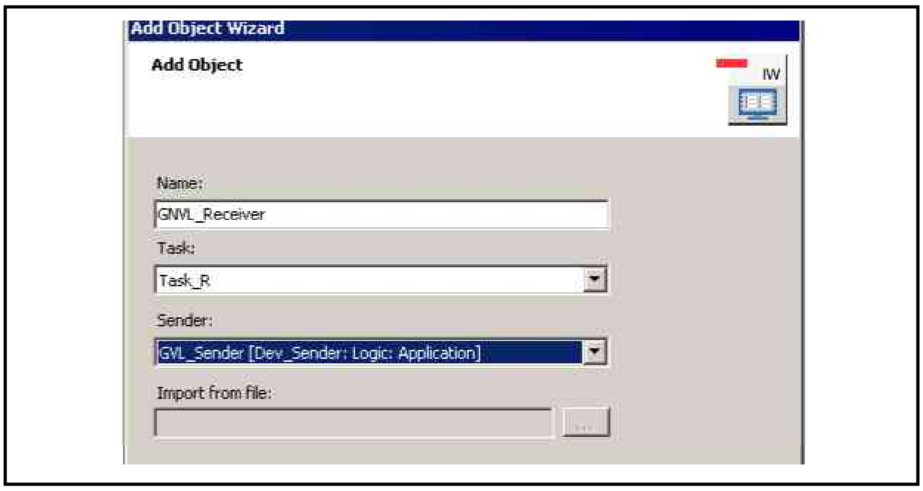
  This way, a point-to-point transmission can be executed between sender and receiver.
- The transmission option from the sender to **several** receivers is ### in preparation ###.

  Here, 255.255.255.255 has to be entered as broadcast address.

- For details, see Network Properties, page 246,

3. Creating a global network variable list in the receiver:

   Highlight the "Application" node in the "Dev_Receiver" control. In the context menu, select **Add** ▸ **Global Network Variable List** to open the "Add Object" dialog.

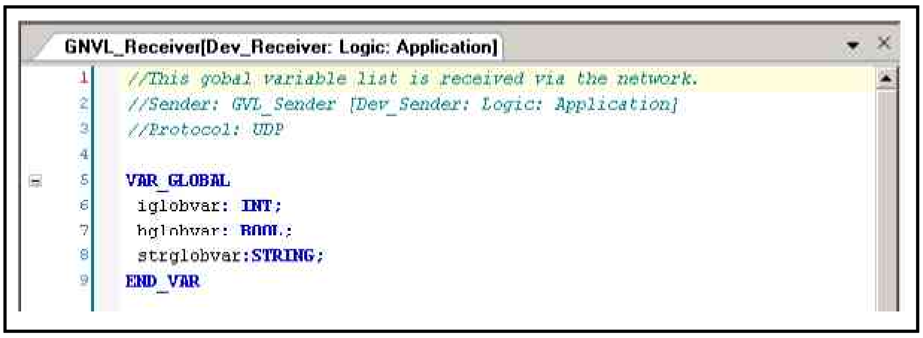

Add Object Wizard
Add Object

Name: GNVL_Receiver
Task: Task_R
Sender: GVL_Sender [Dev_Sender: Logic: Application]
Import from file:

Fig.2-30: Creating a GNVL in the receiving device

Enter the name "GNVL_Receiver". In the "Sender" field there is a selection list of all of the GVL objects currently available in the project with network properties. This example only includes "GVL_Sender". In the "Task" selection list choose "Task_R" in the "Dev_Receiver" control as defined above. Click on "Finish" to confirm your entries.

Double-click on "GNVL_Receiver" to open the "GNVL" editor. This GNVL automatically contains the same variable declarations as "GVL_Sender"; see the following figure.



GNVL_Receiver[Dev_Receiver: Logic: Application]

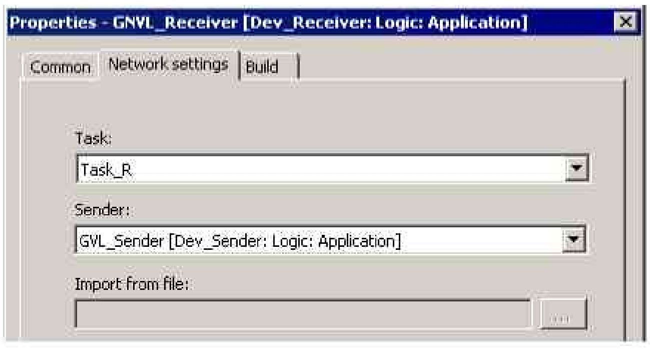
```
//This gobal variable list is received via the network.
//Sender: GVL_Sender [Dev_Sender: Logic: Application]
//Protocol: UDP

VAR_GLOBAL
 iglobvar: INT;
 bglobvar: BOOL;
 strglobvar:STRING;
END_VAR
```

Fig.2-31: GNVL_Receiver contains the variable declarations of GVL_Sender

4. Check or change the network settings of the global network variable list:

   Highlight "GNVL_Receiver" in the Project Explorer and use **Properties...** in the context menu to open the "Properties" dialog. Open the "Network variables" tab and check your entries; see the following figure.



Properties - GNVL_Receiver [Dev_Receiver: Logic: Application]
Common
Network settings
Build

Task: Task_R

Sender: GVL_Sender [Dev_Sender: Logic: Application]

Import from file:

Fig.2-32: GNVL network settings

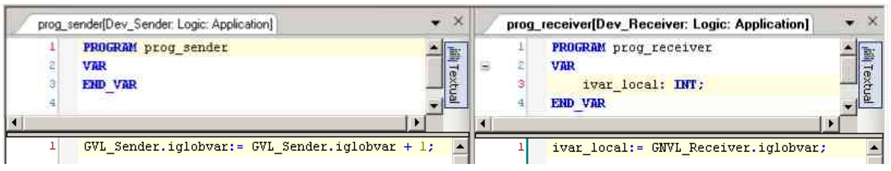
   If necessary, change your entries using the respective selection list and confirm them with "OK".

   - For details, see Network Properties, page 248,

5. Test the network variable exchange:

   In order to test a network variable online, perform the following steps:

   - In "prog_sender" of the sender application use the variable "iglob-var" directly.
   - In "prog_rec" of the receiver application use the local copy of the network variable "iglobvar":
   - Connect sender and receiver applications to the network and start the applications. In the online views of the function blocks, observe whether the values of "iglobvar" in the receiver match with those in the sender.



prog_sender[Dev_Sender: Logic: Application]

```
PROGRAM prog_sender
VAR
END_VAR
```

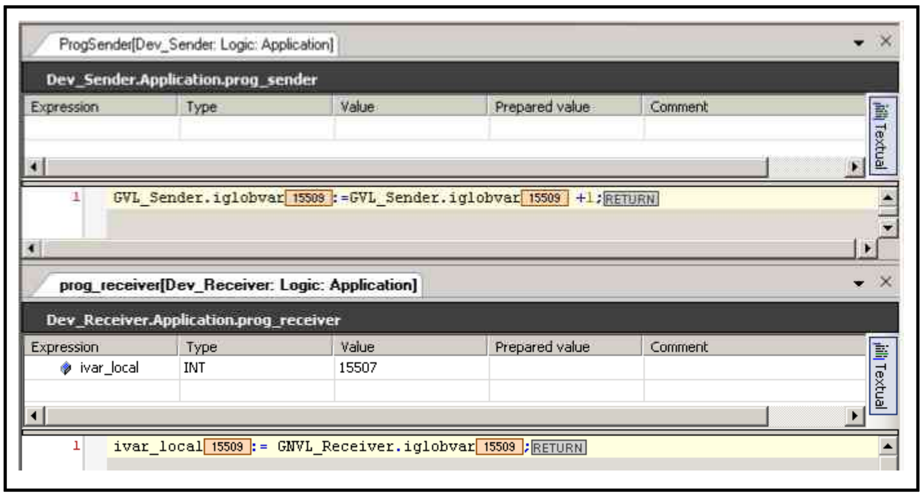
```
GVL_Sender.iglobvar:= GVL_Sender.iglobvar + 1;
```

prog_receiver[Dev_Receiver: Logic: Application]

```
PROGRAM prog_receiver
VAR
    ivar_local: INT;
END_VAR
```

```
ivar_local:= GNVL_Receiver.iglobvar;
```

Fig.2-33: Programming examples, sender end and receiver end


ProgSender[Dev_Sender: Logic: Application]
Dev_Sender.Application.prog_sender
Textual

Expression	Type	Value	Prepared value	Comment

```
GVL_Sender.iglobvar 15509 :=GVL_Sender.iglobvar 15509 +1;RETURN
```

prog_receiver[Dev_Receiver: Logic: Application]
Dev_Receiver.Application.prog_receiver
Textual

Expression	Type	Value	Prepared value	Comment
ivar_local	INT	15507		

```
ivar_local 15509 := GNVL_Receiver.iglobvar 15509 ;RETURN
```

Fig.2-34: Programming examples, sender end and receiver end, transmission running

Network Variables; Data Exchange between IndraLogic 1.x and 2G Controls

It can also be communicated using network variables if the participating controls operate with applications from different versions of the programming system (1.x ↔ 2G).

In this case, the export/import mechanism to create the exactly matching variable lists required in the sender receiver project cannot be used.

This is caused by the differing information in the 1.x and 2G variable export files (*.exp ↔ *.gvl).

If a reading GNVL is set up in 2G, the respective network parameter configuration has to be present as a *.gvl file that was previously exported from the 2G sender. This information is not present in an *.exp file exported from a 1.x sender.

Possible solution for a network variable exchange between 1.x and 2G applications:

1. Reproduce the 1.x NVL in 2G (Add a GVL with network properties containing the same variable declarations as the 1.x NVL).
2. Export the new GVL to an *.exp file ("Linking with file" properties)

☞ Enable the option "Exclude from build" and you can keep the GVL in the project without getting precompile errors and ambiguous names.

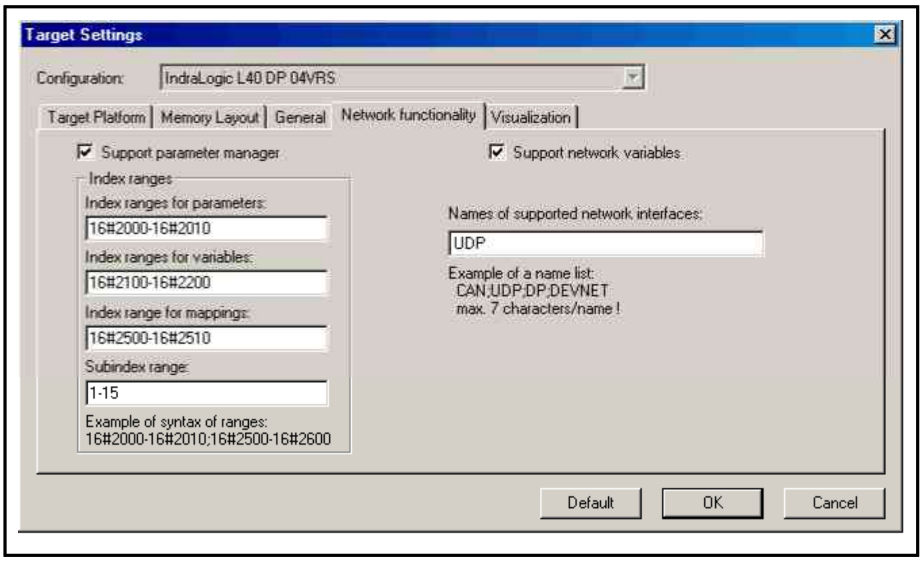
Disable the option if the .exp file has to be created again after the required changes in the GVL.

3. Re-import the list. That means creating a new GNVL using the previously created *.exp file to get a correctly matching variable list in the receiver.

Example:

A prerequisite is that the support of network variables is enabled by a specified control type.

Resources ► **Target Settings** ► **Project Database** ► **Checkout**, then enable "Support network variables".


Target Settings

Configuration: IndraLogic L40 DP 04VRS

Target Platform | Memory Layout | General | Network functionality | Visualization

☑ Support parameter manager

Index ranges

Index ranges for parameters: 16#2000-16#2010

Index ranges for variables: 16#2100-16#2200

Index range for mappings: 16#2500-16#2510

Subindex range: 1-15

Example of syntax of ranges:
16#2000-16#2010;16#2500-16#2600

☑ Support network variables

Names of supported network interfaces: UDP

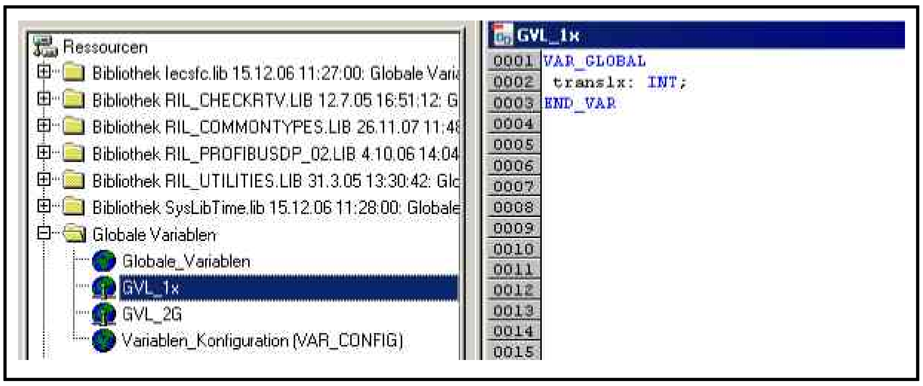
Example of a name list:
CAN;UDP;DP;DEVNET
max. 7 characters/name !

Fig.2-35: Target system setting "Support network variables"

There is one project 1x.pro with one global variable list GVL_1x containing the following declarations:

`VAR_GLOBAL trans1x: INT; END_VAR`.

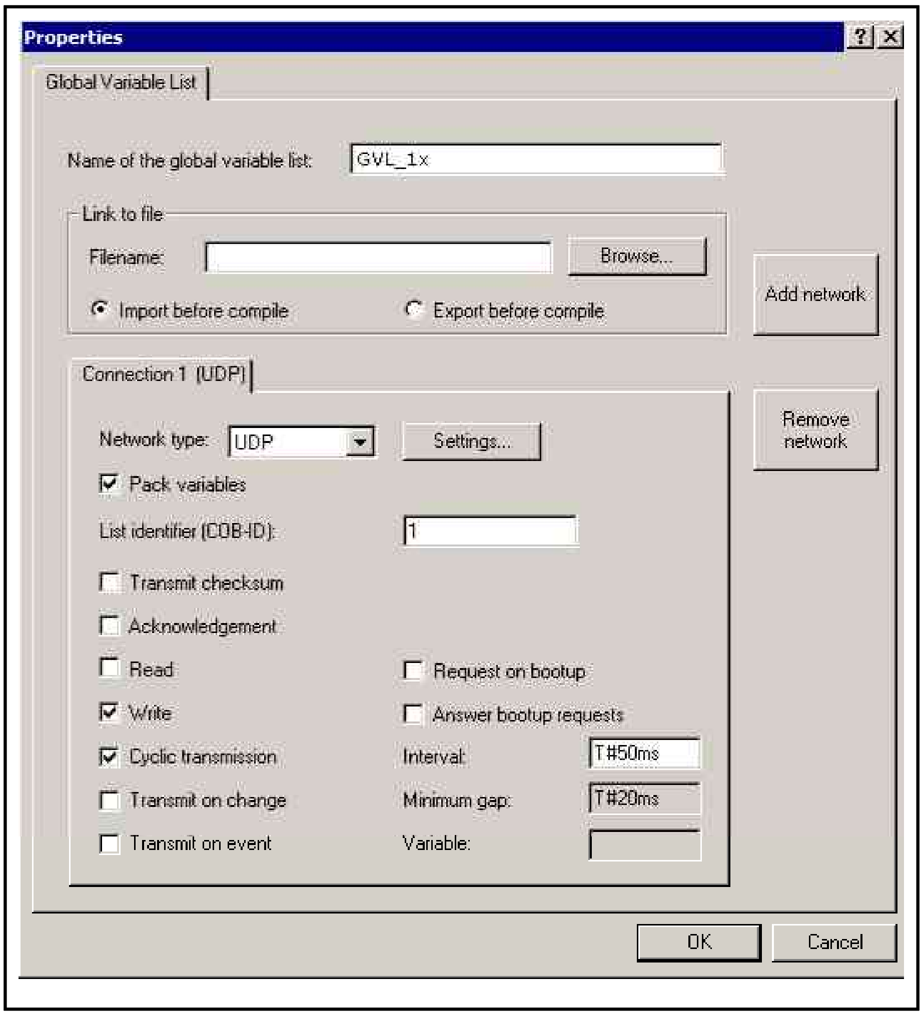
Variable "trans1x" should be possible to be read from a 2G application.


Ressourcen
- Bibliothek Iecsfc.lib 15.12.06 11:27:00: Globale Varia
- Bibliothek RIL_CHECKRTV.LIB 12.7.05 16:51:12: G
- Bibliothek RIL_COMMONTYPES.LIB 26.11.07 11:48
- Bibliothek RIL_PROFIBUSDP_02.LIB 4.10.06 14:04
- Bibliothek RIL_UTILITIES.LIB 31.3.05 13:30:42: Glo
- Bibliothek SysLibTime.lib 15.12.06 11:28:00: Globale
- Globale Variablen
  - Globale_Variablen
  - GVL_1x
  - GVL_2G
  - Variablen_Konfiguration (VAR_CONFIG)

GVL_1x
```
VAR_GLOBAL
  trans1x: INT;
END_VAR
```

Fig.2-36: GVL in the 1x project

The network settings of GVL_1x are configured as follows:



Name of the global variable list: GVL_1x
Link to file
Filename:
(● Import before compile) ( ) Export before compile
Connection 1 (UDP)
Network type: UDP
☑ Pack variables
List identifier (COB-ID): 1
☐ Transmit checksum
☐ Acknowledgement
☐ Read
☐ Request on bootup
☑ Write
☐ Answer bootup requests
☑ Cyclic transmission
Interval: T#50ms
☐ Transmit on change
Minimum gap: T#20ms
☐ Transmit on event
Variable:

Fig.2-37: Properties of GVL_1x

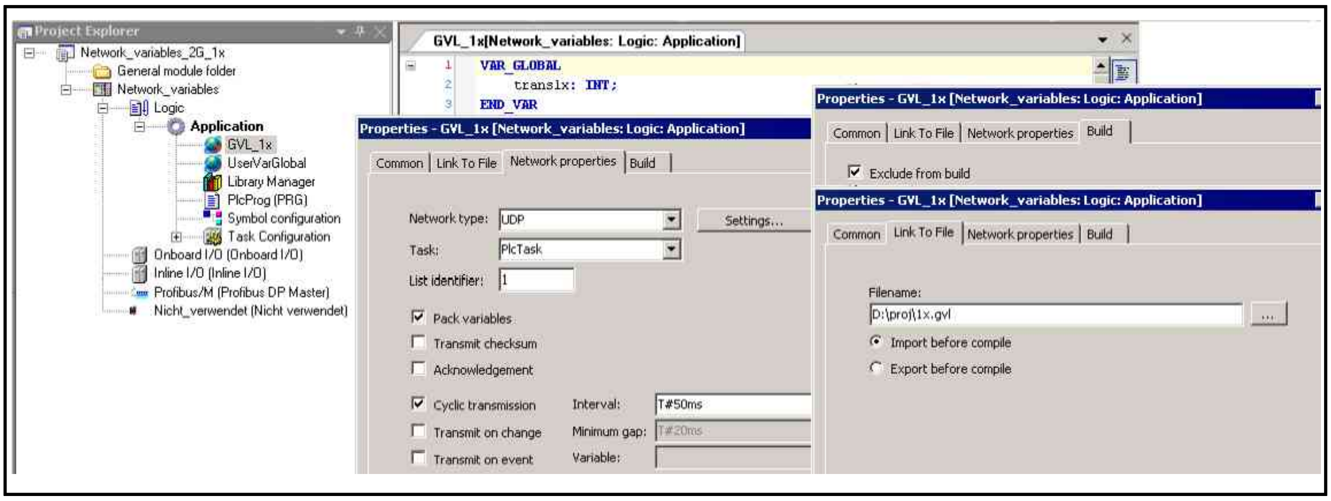
If GVL_1x is exported to an *.exp file, this file contains only the declaration

`VAR_GLOBAL trans1x: INT; END_VAR`

Thus, reproduce GVL_1x in 2G first (see GVL_1x in the figure above):

Add a GVL object with the name "GVL_1x" below an application in a 2G project and proceed with the following steps:

- Set the network properties as defined in 1x.pro
- Specify an export file "1x.gvl" in the "Link to file" properties
- Recommendation: Set the option "Exclude from build" (for details refer to Network properties, page 246)
- Compile the 2G project to generate a 1x.gvl file (contains then variable definitions + configuration data!)



Project Explorer
- Network_variables_2G_1x
  - General module folder
  - Network_variables
    - Logic
      - Application
        - GVL_1x
        - UserVarGlobal
        - Library Manager
        - PlcProg (PRG)
        - Symbol configuration
        - Task Configuration
    - Onboard I/O (Onboard I/O)
    - Inline I/O (Inline I/O)
    - Profibus/M (Profibus DP Master)
    - Nicht_verwendet (Nicht verwendet)

GVL_1x[Network_variables: Logic: Application]

```
VAR_GLOBAL
    trans1x: INT;
END_VAR
```

Properties - GVL_1x [Network_variables: Logic: Application]
Common | Link To File | Network properties | Build

Network type: UDP
Settings...
Task: PlcTask
List identifier: 1
[✓] Pack variables
[ ] Transmit checksum
[ ] Acknowledgement
[✓] Cyclic transmission Interval: T#50ms
[ ] Transmit on change Minimum gap: T#20ms
[ ] Transmit on event Variable:

Properties - GVL_1x [Network_variables: Logic: Application]
Common | Link To File | Network properties | Build

[✓] Exclude from build

Properties - GVL_1x [Network_variables: Logic: Application]
Common | Link To File | Network properties | Build

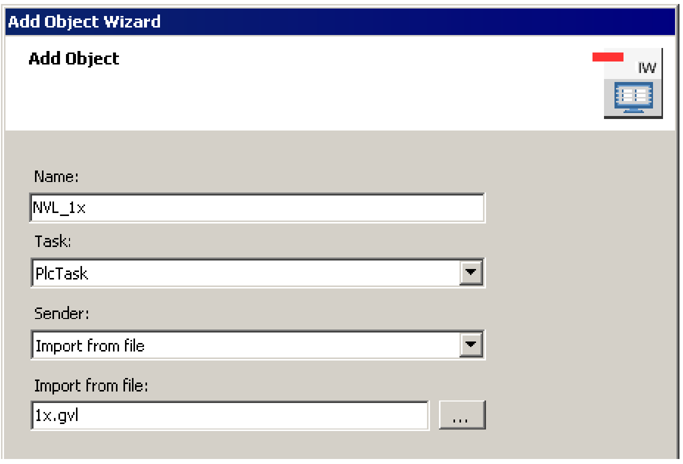
Filename: D:\proj\1x.gvl
(•) Import before compile
( ) Export before compile

Fig.2-38: Reproduce the GVL in 2G

```
<GVL>
  <Declarations><![CDATA[VAR_GLOBAL▯    trans1x: INT;▯
END_VAR]]></Declarations>
  <NetvarSettings Protocol="UDP">
    <ListIdentifier>1</ListIdentifier>
    <Pack>True</Pack>
    <Checksum>False</Checksum>
    <Acknowledge>False</Acknowledge>
    <CyclicTransmission>True</CyclicTransmission>
    <TransmissionOnChange>False</TransmissionOnChange>
    <TransmissionOnEvent>False</TransmissionOnEvent>
    <Interval>T#50ms</Interval>
    <MinGap>T#20ms</MinGap>
    <EventVariable>
    </EventVariable>
    <ProtocolSettings>
      <ProtocolSetting Name="Port" Value="1202" />
      <ProtocolSetting Name="Broadcast Adr." Value="192.168.101.167" />
    </ProtocolSettings>
  </NetvarSettings>
</GVL>
```

Fig.2-39: Resulting export file "1x.gvl" opened in the text editor

Add a GNVL object (option "Import from file) using the 1x.gvl file. This allows to read the variable "trans1x" from the 1.x control.



Add Object Wizard

Add Object

Name: NVL_1x
Task: PlcTask
Sender: Import from file
Import from file: 1x.gvl

Fig.2-40: GNVL in 2G project

If the 1.x project as well as the 2G application run in the same network, the 2G application can read the variable "trans1x" from the project 1x.pro.

2.11 Code Generation and Online Change

Machine code is first generated when the application, page 66, is loaded to the control (PLC).

At each download, the compilation log containing the code and identification of each loaded application is saved as file "IndraLogic.<DeviceName>.<application ID>.compileinfo" in the same directory as the project. The compilation log is deleted when executing the command Clear, page 125, or Clear All, page 125,.

☞ Note that no machine code is generated if the project is compiled with Create commands, page 124,.

This compilation process checks for syntax errors in the project. These are output in the message box.

⚠ CAUTION **Th online change modifies the running application program and causes a restart.**

Ensure that the new application code causes the desired behavior of the controlled system. Depending on the system controlled, damages at the system and workpieces can result or the health and life of people can be put at risk.

☞ *Additional notes:*

1. If an online change is made, the program code may not be as it was before the complete initialization, since the machine keeps its status.
2. Pointer variables retain their value from the last cycle. If a pointer points to a variable that changed its size due to the online change, the value is not provided correctly anymore. Ensure that pointer variables are re-assigned in every cycle.

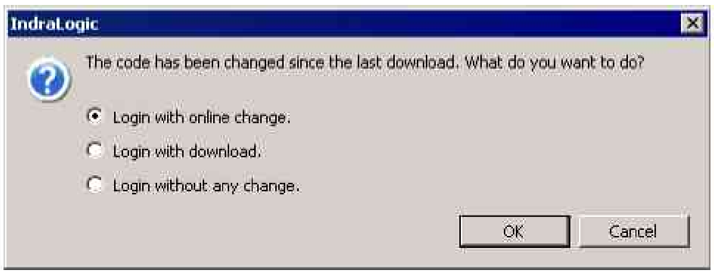
Online change

If the application that is currently running on the control was changed since the last download in the programming system, only the modified project objects are loaded to the control during online change while the program continues running there.

There are two ways to perform an online change:

1. As soon as you try to log in again with a changed application program, a dialog appears prompting what you would like to do. Select from the following three options:
   - Login with online change.
   - Login with download.
   - Login without any change.

   Select the option "Login with online change.".


IndraLogic

The code has been changed since the last download. What do you want to do?

[✓ Login with online change.]
Login with download.
Login without any change.

Fig.2-41: Selection dialog for code download

Confirm with "OK". All changed objects are now loaded and displayed immediately in the online view (monitoring) of the respective object.

Select the "Login with download" menu item and the entire project is loaded to the control.

Select the "Login without any change" menu item and the program on the control continues to run and the new changes are not loaded. Afterwards, download (<application>) explicitly. That download either reloads the entire project or the dialog described above appears again at next login.

2. If you are already logged in and the project on the control is not updated, you can explicitly perform an online change. In the main menu, select **Debug** ▸ **Online Change** to perform an online change.

   A dialog appears prompting if you really want to perform an online change.

   To carry out the online change, click "Yes".

Information on the online change can also be found in "Online Change", page 133,.

Note that an online change in a changed project for an application is no longer possible after a cleanup(commands: Clear All, page 125, "Clear", page 125).

In this case, information on the objects changed since the most recent download is deleted. This means that only the entire project can be reloaded.

Note the following before performing online change:

- Is the modified code free of errors?
- Application-specific initializations (reference motion, etc.) are not executed, since the machine retains its status. Can the new program code really work without re-initialization?
- Pointer variables retain their value from the last cycle. If it is pointed to a variable that changed in size, the value is no longer correct. For this reason, ensure that pointer variables are re-assigned in every cycle.
- If the active step in an SFC chart is removed, the chart remains inactive.

Boot application (boot project)

At each download, the active application is automatically saved as a file called <Application>.app in the target system directory. Click on **De-**

bug ▸ Generate Boot Application to save the boot application in a file even in offline mode.

A boot application is started automatically when the control is started. To do this, the application project on the control has to be available in a file <ProjectName>.app. To create the file, go to **Debug ▸ Generate Boot Application**.

2.12 Monitoring

In online mode, there is a variety of possibilities to display the current values of the variables of an object on the control:

- "Inline monitoring" in the implementation editor of an object.

  Details can be found in the description of the respective editor.
- "Online view of the declaration editor" of an object.

  For details, refer to the declaration editor, page 326.
- "Object-independent monitoring lists"

  For more details, refer to Monitoring Window, page 499.
- "Trace curves"

  Recording and display of variable values on the control. For details, refer to Trace Functionality, page 431.
- "Recipes"

  User-defined variable set to set and monitor these variables on the control. See Recipe Management, page 382.

2.13 Debugging/Troubleshooting

To investigate programming errors, the debugging functions in IndraLogic 2G can be used.

Note the option of an application in the simulation mode, page 183, that is without necessary connection to a real target device.

Breakpoints can be set at certain positions in the program to force an execution stop. Certain conditions, specifically which task(s) are affected and in which cycle intervals the breakpoint is to be effective, can be defined for each breakpoint.

Single step processing enables the program to run in controlled steps.

At each stop, defined by the step marks and breakpoints, the respective variables can be investigated.

Breakpoints

A breakpoint set in an application program causes a stop in the execution of the program. The possible breakpoint positions depend on the respective program editor. There is always a breakpoint position at the end of the POU.

A description of the command for handling breakpoints can be found in "Breakpoints", page 126,. An important tool is the "Breakpoints" dialog, page 136 in which all defined breakpoints are listed and in which breakpoints can be added, deleted or modified.

Conditional breakpoints. The stop at the breakpoint can depend on the task currently executed or the number of the cycle currently running.

Breakpoint icons

● Breakpoint activated

○ Breakpoint deactivated

➲ Stop at the breakpoint in online mode

Single step processing

Single step processing (stepping) enables a controlled execution of the application program, e.g. for the purpose of troubleshooting. Repeated pressing of <Alt>+<F12> allows to jump from instruction to instruction. However, called function blocks can also be skipped.

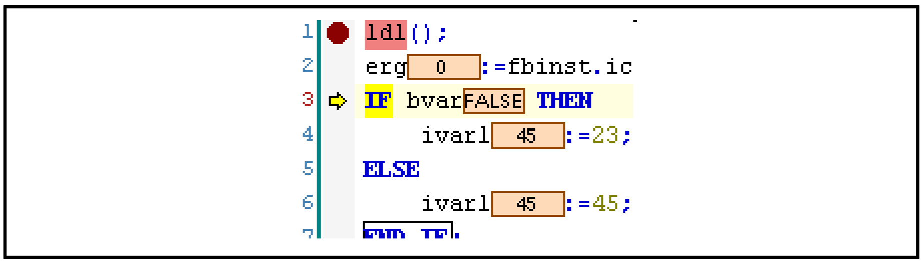
What's new compared to IndraLogic 1.x

- The instruction to be executed as next instruction can be explicitly defined. To do this, click on **Debug** ▸ **Specify Next Instruction** in the main menu.
- The next execution stop can be determined by placing the mouse pointer at the desired position. To do this, click on **Debug** ▸ **Execute to cursor** in the main menu.
- "Execute to Return" causes a backward step to the last call. To do this, click on **Debug** ▸ **Execute to Return** in the main menu.

A description of the stepping commands can be found in "Breakpoints", page 126,.

Icon for single step processing (stepping):⇨

The current position during stepping is displayed with a yellow arrow in front of the line and yellow shadowing of the related operation.



```
ldl();
erg   0   :=fbinst.ic
IF bvarFALSE THEN
    ivarl   45   :=23;
ELSE
    ivarl   45   :=45;
```

Fig.2-42: From the breakpoint, the command "Single step" is used to jump to the next step

2.14 Printing

The view in the currently active editor can be printed using the "Print" function. To do this, click on **File** ▸ **Print** in the main menu. Note the alternative possibility for generating a "Documentation" of selected objects in the project in a defined layout and with a table of contents. A detailed description about "printing" can be found in the IndraWorks documentation.

2.15 Visualization

Information on the visualization in IndraLogic 2G and the visualization editor can be found in "Visualization," page 625, and "Visualization editor", page 445.

2.16 Library Management

2.16.1 Library Management, Overview

Libraries can provide functions, function blocks, data types, global variables and even visualizations which can then be used in the project.

The default extension for a library file is ".library" in contrast to ".lib" which was used for files in IndraLogic V1.x and previous versions. Encrypted libraries have the extension "*.compiled-library".

The management of libraries in a project occurs in the library manager. The previous installation on the system is performed using the "Library Repository" dialog.

The project functions for a global and local "find" and "replace" can also be used for libraries that are not encrypted.

See the following general information on:

- Installation on the System and Integration into a Project, page 84
- Referenced Libraries, page 85
- Library Versions, page 85
- Unique Access to Library Function Blocks (Namespace), page 86
- Creating a Library in IndraWorks, page 191,
- IndraLogic 1.x Libraries, page 86
- External and Internal Libraries or Library Function Blocks, Late Linking, page 87

2.16.2 Installation on the System and Integration (Linking) into a Project

- Libraries can be managed on the local system in various "repositories" (directories, storage locations). Before a library can be integrated into a project, it has to be installed on the local system in a repository.

  This is done in the Library Repository dialog, page 185, in IndraLogic.
- A prerequisite for the installation is that the library information library information, page 371, of a library project has a title, version information and the name of the vendor (company).

  As an option, a category designation can be entered that can later be used in the library manager for sorting.
- If there is no category assignment in the library information, the library automatically belongs in the "Other" category. If other library categories in addition to this default category are to be used in IndraLogic libraries created in 2G, these are defined in one or more external XML file(s) "*.libcat.xml" that can also be extended and created again. Such a file can then be called in the "Library Information" dialog to select a category. For further information on the categories, refer to Creating a "Library" or "Compiled Library" in the IndraWorks Environment, page 191
- Libraries are integrated into a project with the Library manager, page 367,. In a "default project", it is automatically assigned to the default device first. But it can also be added explicitly in Project Explorer, page 63, (below a device or an application) or globally in the "General module" folder. This is done, as for other objects, using the Add Object dialog, page 234. To do this, highlight the application and select **Add ▸ Library manager** in the context menu. Libraries that are integrated into a library are also displayed with a preset in the library manager. However, "hidden libraries" are also possible; see also Referenced Libraries, page 85.
- If the library is not in encrypted and the ".library*" file is present instead, the library POUs listed in the library manager can be opened by double-clicking on the respective entry.
- If a library function in the project is addressed, the libraries and repositories are searched in the sequence in which they are listed in the "Library Repository" dialog; see also Unique Access to Library Function Blocks (Namespace), page 86.

2.16.3 Referenced Libraries

- A library can link other libraries (referenced libraries) where the nesting can be as deep as desired.

  If such a "father" library is linked to the library manager of the global "General module" folder of a project, both the father library and the libraries it references are available in **all** applications of the project.

  If such a "father" library is linked to the library manager of an application, both the father library and the libraries it references are available **in exactly these** applications of the project.

If, for example, the library "RIL_Utilities" references the library "Util" and "Util" contains the function block "BLINK", an instance "bk1" of "BLINK" has to be declared as follows:

```
bk1: RIL_UTILITIES.UTIL.BLINK;
```

- When creating a library project that references others projects, it can be specified in the Properties, page 230, of each linked library how it has to act later when it is linked to a project via the " father" library:
  1. Its visibility in the Library Manager, page 367, indented below the "father" library, can be disabled. This way, "hidden libraries" can be provided in a project.
  2. If a pure "container" library is generated - in other words, a library that does not define any function blocks itself but instead only references other libraries - later access to its function blocks can be simplified.

     When a "container" library is linked to a project, a whole set of libraries is linked along with it.

     In this case, it is possible to simplify the access to the function blocks of these libraries by defining them as "top level" libraries. Then, when accessing the function blocks, the namespace for the libraries can be omitted.

     To do this, use the "Publish..." option in the library properties. However, this option should only be used when creating a container library project!
- See also Library Management, page 83.

2.16.4 Library Versions

- Several versions of a library can be installed simultaneously on the system.
- Several versions of a library can be simultaneously linked to a project. The following are clearly specifies on which version an application accesses in this case:
  – If several versions are located on the same level in the Library manager, it depends on the definition in the Library properties, page 230, which version is to be used (a certain one or always the newest one).
  – If several versions are located on different levels (which can be the case with referenced libraries, page 85), unique access to library function blocks is achieved by entering the corresponding namespace as described in the following.

2.16.5 Unique Access to Library Function Blocks (Namespace)

- Basically, the following applies:

  If several function blocks with the same name are available in the project, access to a function block component has to be unique or compiler errors result. This applies to project-local function blocks and to function blocks available in linked, referenced libraries. In such cases, the unambiguousness is achieved by adding the namespace in front of the function block name.
- The default namespace of a library is defined in the Library Properties, page 230,.

  If it is not explicitly defined, the library name is automatically used. However, when creating a library project, another default namespace can be entered into the "Properties" dialog. Later, when a library is already linked to the library manager of a project, the namespace can also be changed locally - also in the "Properties" dialog.
- In the following examples, the "namespace" of the library "Lib1" is added to the library properties with "Lib1". In the right column, there are the namespaces for unique accesses to the variable "var 1" defined in the function block "module1" and in the function block "POU1".

	Variable "var1" defined in the positions (1) to (5) in the project:	**Unique access to "var1" using the corresponding namespace information...**
(1)	In the library "Lib1" in the global library manager in the "General module" folder	"Lib1.module1.var1"
(2)	In the library "Lib1" in the library manager below an application "App1" of a control "Dev1"	"Dev1.App1.Lib1.module1.var1"
(3)	In the library "Lib1" linked to the library "F_Lib" (referenced) in the global library manager in the "General module" folder	Presets: (Option "Publish..." is disabled in the library properties of Lib1 when "Lib1" is added to "F_Lib"): "F_Lib.Lib1.module1.var1" If the option "Publish..." was activated, "module1" would be treated as a component of a library linked at top level. Then, access without entering the namespace of the "father" library "F_Lib" is normally possible: "Lib1.module1.var1" or "module1.var1"). In the present example, however, this leads to a compiler error because the call is no longer unique; see points (1) and (4).
(4)	In the object "module1" that is defined in the "General module" folder	"module1.var1"
(5)	In the object "POU1" that is defined in the "General module" folder	"POU1.var1"

Fig.2-43: Namespaces

2.16.6 IndraLogic 1.x Libraries

- Libraries that were created with IndraLogic 1.x (*.lib) and earlier versions continue to be supported.
- An old library project (*.lib) can be opened directly in IndraLogic 2G and can be converted into an "IndraLogic 2G library" (*.library).

- If an old project that references old libraries is opened, it can be selected whether these references are to be retained, replaced or deleted. If they are to be retained, the affected libraries are converted into the new format and are automatically installed in the system library repository.

  If they do not contain the necessary library information, page 371,, these can be immediately added.

  The model (mapping) by which an old library was once handled during conversion of an old project can be saved in the project options so that the same library does not need to be explicitly handled each time in future project conversions.
- A description of the procedure for converting projects and libraries can be found in Data Transfer, page 115,.

2.16.7 External and Internal Libraries or Library Function Blocks, Late Linking

- An "external library", in contrast to an internal library (IndraLogic library project), is a library file that is programmed outside of IndraLogic in another programming language, e.g. C. It has to be present on the target system and is only linked if the application is running there.
- As in IndraLogic 1.x, it is also possible to link an IndraLogic library as an external library later on, i.e. when the application is first operated on the runtime system. In addition, it is also possible now to define the late linking individually for every library function block.

  For this purpose, the property in the object properties of one or all function blocks can be enabled ("External Implementation", page 243).